AD-A115 552 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
SIMULATION AND ANALYSIS OF THE AFLC BULK DATA NETWORK USING ABS--ETC(U)
DEC 81 S D STEWART
UNCLASSIFIED AFIT/GCS/EE/81D-15 NL

1 OF 2
A
115552

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/EE/81D-15

SIMULATION AND ANALYSIS OF THE AFLC BULK DATA NETWORK USING ABSTRACT DATA TYPES

THESIS

Stephen D. Stewart
GCS-81D

AFIT/GCS/EE/81D-15

SIMULATION AND ANALYSIS OF THE AFLC BULK DATA NETWORK USING ABSTRACT DATA TYPES

by

Stephen D. Stewart

Thesis

Prepared in Partial Fulfillment of the
Requirement for the Degree of
Master of Science

Graduate Computer Systems
December, 1981

School of Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base,
Ohio

Preface

The purpose of this study was to develop and validate a computer simulation of the AFLC Bulk Data Network using abstract data types. The model was used to study alternative network designs. Previous network simulations, which were written in QGERT, required extensive core memory and were difficult to use. Certain features of the BDN, such as its capability to transmit a tape to many sites simultaneously, would have been very difficult to program in a special purpose simulation language. However, using PASCAL, I had had no trouble modeling these features. By representing the basic network components by abstract data structures, I was able to create my own network modeling language. The program listed in Appendix C should benefit anyone attempting a similar project.

I would like to express my thanks to three faculty members, Lt. Col. James Rutledge, Dr. Gary Lamont and my advisor, Major Walter Seward, for their interest, advice and encouragement during this thesis effort. Finally, I would like to thank my wife, Nancy, and my son, Michael, whose love and understanding endured through this graduate program.

Contents

List of Figures

List of Tables

Abstract

The AFLC Bulk Data Network (BDN) is a packet switching network which electronically transfers bulk computer data (magnetic tapes) between the six AFLC bases. The network operates at "full" capacity for extended periods of time at the beginning of each month. This results in undesirable delays in transferring data between the sites. To remedy the problem, AFCCPC/SKDAR plans to add an additional circuit to the network and increase the transmission capacity of another circuit. A computer simulation was developed and validated to analyze this and other proposals formulated to improve network performance.

The simulation makes extensive use of abstract data types to represent key network components. The abstract data structures provide a set of functional building blocks which are linked together to model a variety of network configurations. The program incorporates full node to node link and tape to tape message protocols. The model can be used to study both interactive and batch traffic, using one of three different routing algorithms.

SIMULATION AND ANALYSIS OF THE AFLC BULK DATA NETWORK USING ABSTRACT DATA TYPES

I. Introduction

The Bulk Data Network (BDN) is a packet switching network which is used by the Air Force Logistics Command (AFLC) to move magnetic computer tapes between the six AFLC bases located in widely separated areas of the country (Figure 1). According to the system manager, Mr. Robert Norman, AFCCPC/SKDAR (9), the network currently consists of six UNIVAC 418-II host computers, one per site, which are connected by seven full duplex 4800 baud circuits. A full duplex 4800 baud circuit is simply a telephone line which is capable of transmitting 4800 discrete bits of information per second in each direction simultaneously. A half duplex circuit is a telephone line which can transmit information in only one direction at a time.

The magnetic tapes are read into the computer input buffer, one block at a time. The block is then broken into individual packets of 174 characters each. The packets are placed in a queue for transmission to their destination. The route a packet takes depends on the current loading of the network. Each site transmits a status packet, describing its current loading and connectivity knowledge, to its

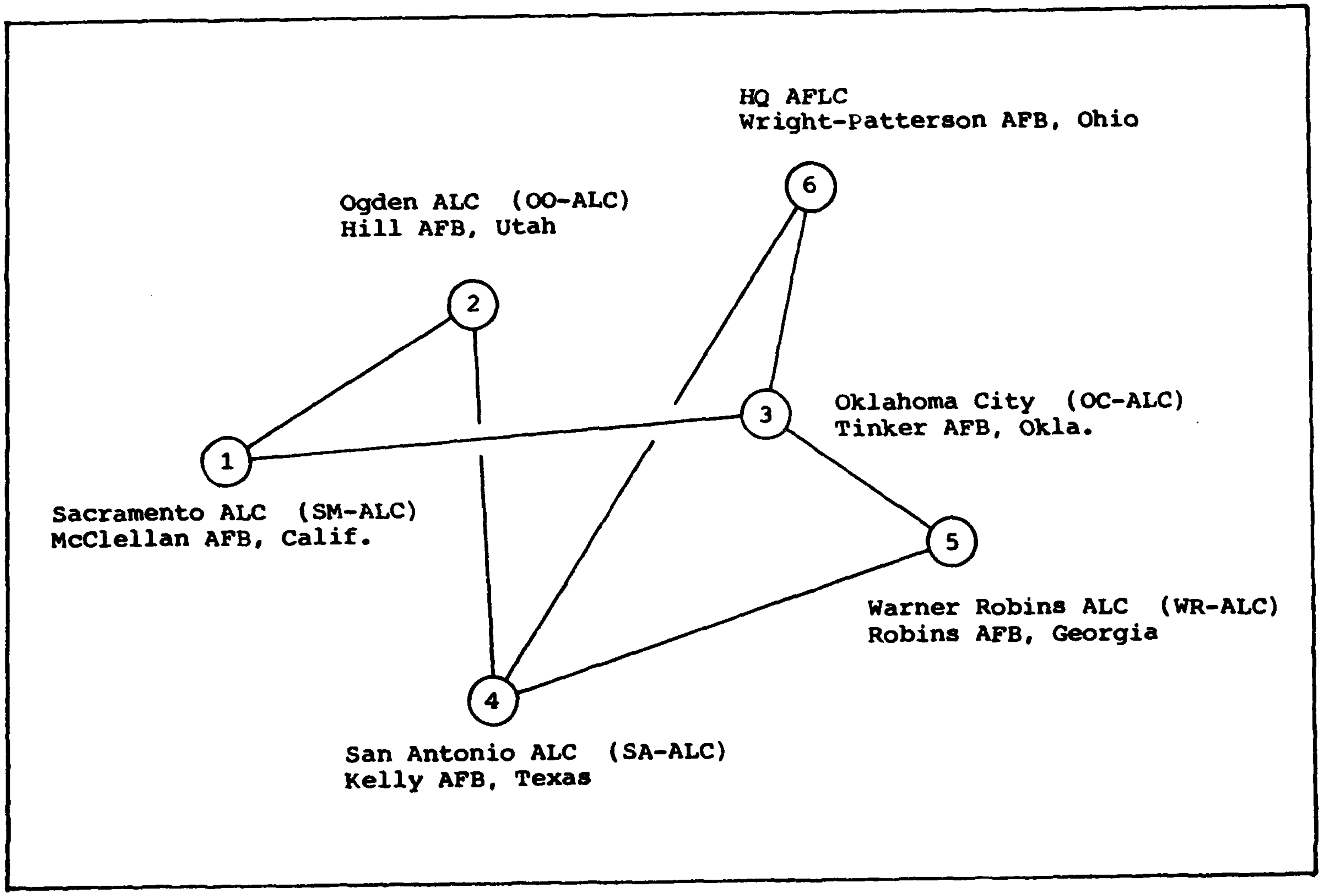

Figure 1. AFLC Bulk Data Network.

neighboring sites every 12 seconds. This information is used to determine the quickest path for each packet. When the line is free, a packet is sent on its way. The packets are collected and reassembled into blocks at the final destination. Finally, the blocks are written onto an output tape. Each node can handle up to five output tapes and four input tapes at one time.

The BDN currently transmits 1700-1800 magnetic tapes per month. Normal daily tapes consist of 1000 blocks x 3000 characters per block. However, end of week and end of month tapes range from one to three reels in length. One reel consists of 6000 blocks of 3000 characters each. End of month reports usually keep the network saturated from the third to tenth of the month. This results in undesirable delays in transferring data between the bases. In an effort to resolve the problem, SKDAR plans to add a new 9600 baud circuit between Hill Air Force Base (AFB), Utah, and Wright-Patterson AFB, Ohio, and upgrade the 4800 baud circuit between McClellan AFB, California, and Hill AFB, Utah, to 9600 baud.

In addition, HQ AFLC/LO has asked SKDAR to determine if the BDN could handle ten interactive terminals, five at Robins AFB, Georgia, and five at McClellan AFB, California, connected to a computer at Headquarters AFLC in Ohio. AFLC estimates that each terminal will generate 1000 packets per day. These additions will certainly increase the workload on the network, but because of the complex interaction of the various components, no quantitative evaluation of the proposal has been made.

Objective

The objective of this study was to develop a computer simulation of the network which could be used to evaluate the proposals.

Features

The BDN is characterized by its ability to handle classified (secret) tapes, data suppression techniques, multi-destination tape transmission, and unique packet routing algorithm. All data packets are encrypted at the source and decrypted at the destination to prevent the information from being compromised in the event the packet is electronically intercepted during the transmission process. The BDN can currently handle routine, confidential and secret information. SKDAR has requested certification to transmit top secret tapes. In addition to the encryption/decryption procedure, a data suppression algorithm is used to reduce the actual number of characters transmitted between sites. The data suppression algorithm removes duplicate character strings prior to transmission at the source and replaces them at the destination. Data blocks are normally compressed to 10 to 50% of the original size. The actual figure varies from tape to tape and block to block, depending on the information contained. Each output tape can be transmitted to one or all sites simultaneously. Each data packet is addressed to all destinations. As a packet passes through a node, the host computer processes the message and then transmits the packet to the next site. Approximately 20% of the tapes are multi-addressed.

In most networks, the routing decision is made when a packet first enters a node. The packet is then placed in a queue for a specific output circuit. In the BDN, however, all incoming packets are placed in a general job queue and the routing decision is made at the time of transmission. When a circuit is free, the job queue is searched to find the first packet that can be output over the free circuit.

Approach

The study was conducted in four stages. A detailed functional analysis of the BDN was made, the simulation model was developed, the model was implemented and tested and finally, the model was used to study the proposed changes.

The functional analysis was performed using information provided by SKDAR. The objective of the analysis was to clearly identify the network data flow and control functions. The completed functional analysis served as the basis for the development of a computer simulation model.

The simulaaion program, which was written in PASCAL, is a crude network modeling language. The program defines four network building blocks, nodes, circuits, tape drives and terminals, which are used to construct the model network. The network architecture and topology, which consist of the number of nodes, tapes, connectivity, transmission rates, etc., are defined by the user through input variables. The model simulates the operation of the network for a specified time and then prints out a number of block and packet statistics. The program was implemented on the AFIT CDC Cyber computer and tested by modeling the current network and running a sample workload with known results.

After the model was validated, the program was used to model the upgraded network. Six sample workloads, with varying numbers of tapes and message lengths (number of packets per blocks), were run on each configuration to test them under light, medium and heavy loading conditions. The total number of tape blocks transmitted and maximum interactive packet delays were used as performance measures to compare the runs.

Assumptions

The following assumptions were made in developing the model:

1. The generation of interactive data packets from a terminal is a Poisson process. This assumption is substantiated by previous studies (12, 13).

2. The connectivity of the network is not allowed to change during a simulation run (i.e., circuits will not enter or drop out of the network). The effect of connectivity changes can be studied using different simulation runs.

3. All data packets contain a full 174 characters. In the actual network, the last packet in a block can contain anywhere from 31 to 174 characters depending on the length of the data block and the results of the data suppression routine.

4. Terminal data packets are also 174 characters each. The terminal packets will be compressed using the data suppression algorithm and therefore, the actual packet size is unknown.

Sequence of Presentation

The next chapter discusses several network modeling techniques and explains why computer simulation was selected. Chapter III and IV provide a detailed functional analysis of the current network and the proposed changes. The simulation model design is outlined in Chapter V and the program implementation and testing is described in Chapter VI. Chapter VII presents the results of the study and Chapter VIII gives the conclusions and recommendations.

II. Network Evaluation Techniques

There are several ways to analyze the performance of a computer communications network. The selection of a particular technique depends upon the performance characteristics to be measured, desired level of detail and available resources. Generally, the more detailed the study, the greater the cost. This chapter examines three basic network performance measures and five common modeling techniques. It concludes with the selection of a technique for evaluating the BDN.

Performance Measures

The three basic performance measures applied to computer communication networks are throughput, message delay and cost. Throughput is the rate at which finished outputs are produced. The BDN has two basic outputs, data blocks for batch traffic and packets for interactive traffic. Packet throughput is not a valid measure of performance for data blocks, because a number of duplicate packet transmissions may be required to complete a block transmission. Therefore, the average packet throughput does not directly correspond to the block throughput. Message delay is the average time for a single message transfer. For convenience, we will define network response to be the reciprocal of message delay (i.e., as the message delay goes down, the

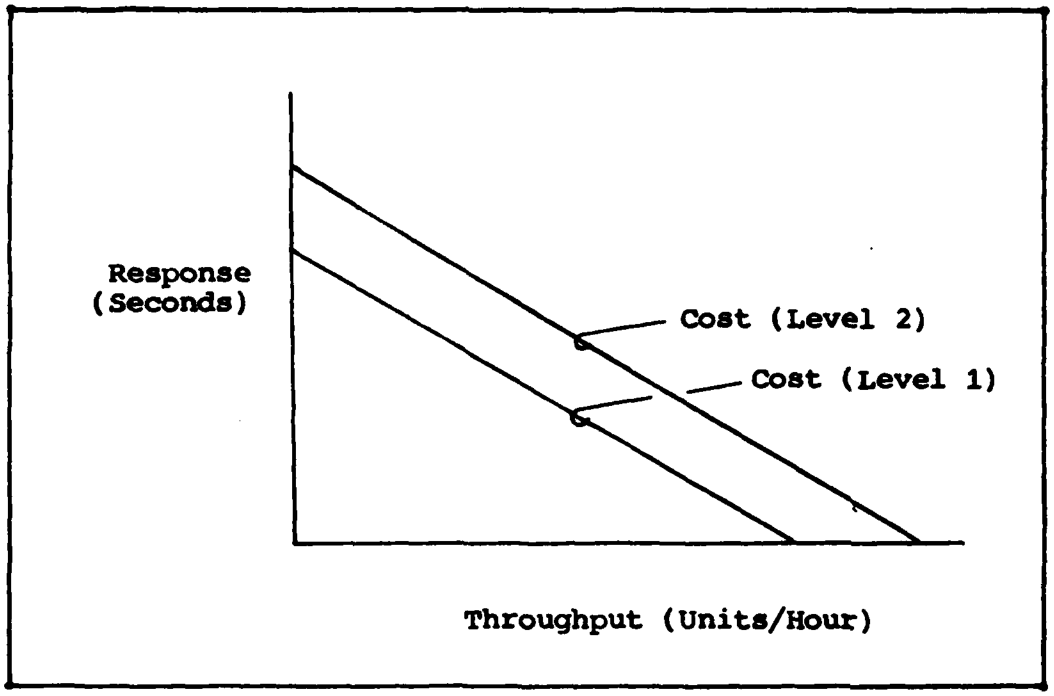

Figure 2. Performance Measures.

network response goes up). The relationship between throughput, response and cost is shown in Figure 2. At a given cost level, additional throughput can be obtained at the expense of network response and vice versa. Maximum throughput is obtained by ensuring that there will always be a packet waiting in the queue when a circuit is ready. Conversely, maximum response is obtained by ensuring that a circuit is idle when the packet arrives. Both throughput and response can be improved by adding resources, but this increases the total cost of the system. Network models are used to study the relationship between cost, response and throughput for a specific network configuration.

Message delay is the primary concern for interactive traffic, because the terminal operator must wait for a response before entering the next transaction. In batch traffic however, individual packet response times are relatively unimportant and network throughput is the main concern. The objective of this study is to obtain an optimal balance between interactive and batch traffic at a fixed cost level.

Modeling Techniques

Seward (11) identified five methods for modeling a computer network: rule of thumb, linear projection, analytical modeling, simulation and benchmarking. These methods vary widely in both cost and complexity. The advantages and disadvantages of each method are described below.

Rules of Thumb. The rule of thumb approach uses generalizations to deduce operating characteristics (e.g., the network loading should never exceed 40%, or if one is good, two are better, etc.). Some of these generalizations are based on valid experience and can be useful, but in general, the method is not completely reliable.

Linear Projection. Linear projection is a technique which assumes that performance characteristics are directly proportional to physical characteristics and that changes to one lead to equal changes in the other. An example would be to assume that doubling the capacity of a circuit will double its throughput. This may or may not be the case, depending on whether or not the transmitting and receiving computers can handle the increased workload. If either node cannot do the job, the real throughput will not double and in some cases, might even go down as a result of overloading the input queues. Linear projection can provide useful estimates when the relationships are linear, but in most cases they are not, and the method is of little use.

Analytical Modeling. Analytical modeling, which includes both deterministic and probabilistic models, is a powerful and relatively inexpensive method for studying a network. These models use complex mathematical representations of the network to predict message delay times and queue lengths. When both the message interarrival and transmission times are constant, a purely deterministic network model can be constructed. When the interarrival or service times are not constant, it may be possible to construct a queueing model of the network.

The specification of a network queueing model requires detailed knowledge of the network's message population (source of the messages), arrival distribution, queue discipline, queue configuration, service discipline and service facility. Queue discipline refers to the way messages enter or reject a queue and their degree of patience once in the queue. For example, do packets enter the rear of the queue? The middle? Can they refuse to enter a queue if it looks crowded? Or, once in the queue, can packets leave before being served? Queue configuration refers to the physical characteristics of the queues. Are they finite? Infinite? Is there a single queue or many queues for each server? The service discipline refers to the order in which messages are transmitted. Examples are First-In/First-Out (FIFO), Last-In/First-Out (LIFO), Shortest-Message-First, etc. Service facility refers to the physical characteristics of the transmission equipment, baud rate, number of channels, etc.

The major disadvantages of queueing models are that they only provide a generalized view of the network and are difficult to formulate for large complex systems. As a result, queueing models cannot be used to study specific network control functions and are usually used in conjunction with other modeling techniques.

Simulation. Computer simulation is a process which involves building a computer program that models the functional characteristics of the network and running a sample workload on the model to see how it performs. Simulation is more expensive than queueing, but it is often the only way to study complex funtional relationships in a large system. Unlike queueing models, simulation can be used to study the impact of different control features, such as the routing algorithm and link protocols.

Simulation is based on a detailed functional description of the network. The functional description is used to both build and verify the simulation model. Once a functional description has been obtained, the program can be developed using either a special purpose simulation language, such as QGERT or SLAM, or a general purpose computer language like FORTRAN or PASCAL. Simulation languages provide built-in functions, such as queues and servers, which can simplify the modeling process. General purpose languages are more flexible and allow the user to model unique network characteristics (12). The completed model must be verified to ensure that it performs the functions specified in the functional description and validated to ensure that the model output represents actual network performance. The model is verified by mapping the program functions on to the functional description. It is validated by running a set of data with known results or comparing the simulation results with those of a queueing model. If valid, the model can be manipulated to study changes to the real network.

Benchmarking. If a similar network exists, a representative workload, called a benchmark, can be run on the similar system to see if the network can handle it. If such a system does not exist, the designer may wish to build a prototype network with only a few nodes and test the benchmark workload on it. A major concern with both the simulation and benchmarking techniques is that the sample workload does not test all possible conditions and a critical condition which the network cannot handle can be overlooked. Building a prototype network is very expensive and would not be done until a network simulation has proven that the basic concepts are feasible.

Model Selection

Simulation was chosen as the primary modeling technique because it provides for a detailed examination of the network control functions. The simulation measures both block throughput and interactive packet delay. Analytical models were used to validate portions of the simulation.

A general purpose language (PASCAL) was chosen to implement the model. The unique features of the BDN, multi-destination packets and a centralized job queue, are not compatible with the special purpose languages; they would require extensive reliance on user functions. A previous model (3) which used QGERT required large amounts of main memory (200K words) and therefore, could not be run interactively. Interactive programs are much easier to develop and use because the user can communicate directly with the computer. QGERT and SLAM require extensive memory because they are written in FORTRAN, which does not allow dynamic memory allocation. Large arrays must be set up at the beginning of the program to accommodate the maximum number of variables allowed, even though only a small number would be used. PASCAL, which provides dynamic memory allocation, (i.e., memory is assigned as it is required) should produce a much smaller program which can be run interactively. In addition, PASCAL's highly structured procedures and data types simplify model development, use and testing.

Summary

The three performance measures used to evaluate network performance are: throughput, message delay and network cost. Individual packet delay is balanced against total throughput to obtain the desired network performance characteristics at a fixed cost. In the BDN, block throughput is the most important measure of tape traffic and message delay is the significant factor in interactive terminal traffic.

There are five ways to model a computer network: rule of thumb, linear projection, analytical modeling, simulation and benchmarking. Of these, computer simulation was chosen to evaluate the BDN, because it provides a means for studying the complex message routing and control functions. The simulation was written in PASCAL to make use of the structured data type and dynamic memory allocation features provided by the language.

The next chapter begins the simulation process by developing a detailed description of the current network. The following chapter provides a similar analysis of the proposed changes.

III. Analysis of the Network

A detailed functional analysis of the network provides a basis for the development of the simulation model. The major characteristics of a communication network are its data flow and network control functions. Softech's Structured Analysis and Design Technique (SADT) (10) was used to analyze the operation of the BDN because of its strong emphasis on identifying both these functions. This chapter begins by describing the SADT technique; the technique is then used to make a functional analysis of the Bulk Data Network.

Structured Analysis and Design Technique

SADT is a top-down diagramming technique which shows the component parts of a system, the interrelationships between them and how they fit into a hierarchic structure. Ross (10) describes a process as an activity which uses mechanisms (system resources) to transform input into outputs, subject to certain *controls*. The SADT Activity *Diagram* (Figure 3) is used to represent this process.

The diagram is made by placing the name of the activity inside a box. Inputs are always shown as arrows going into the left side of the box and outputs as arrows coming out of the right side. The controls are shown as arrows entering the top of the box and mechanisms by arrows entering the bottom.

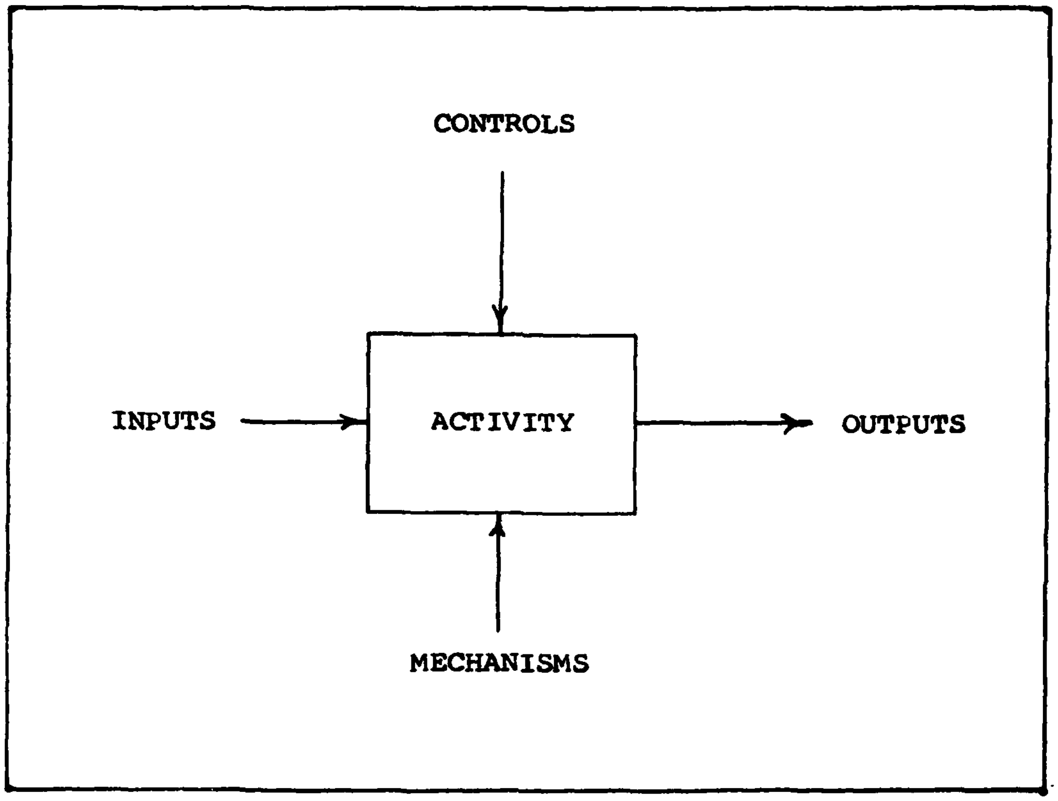

Figure 3. SADT Activity Diagram.

The structured analysis process begins by identifying the processes' main activity, inputs, outputs, mechanisms and external controls. Next, the main activity is decomposed into major subactivities. These activities are shown on a single page using a block for each activity (Figure 4). The boxes are connected by arrows which are labeled to show the relationship between activities. The output from one activity is usually an input to the next activity. However, some outputs feed back as controls to earlier activities; oth outputs leave the system

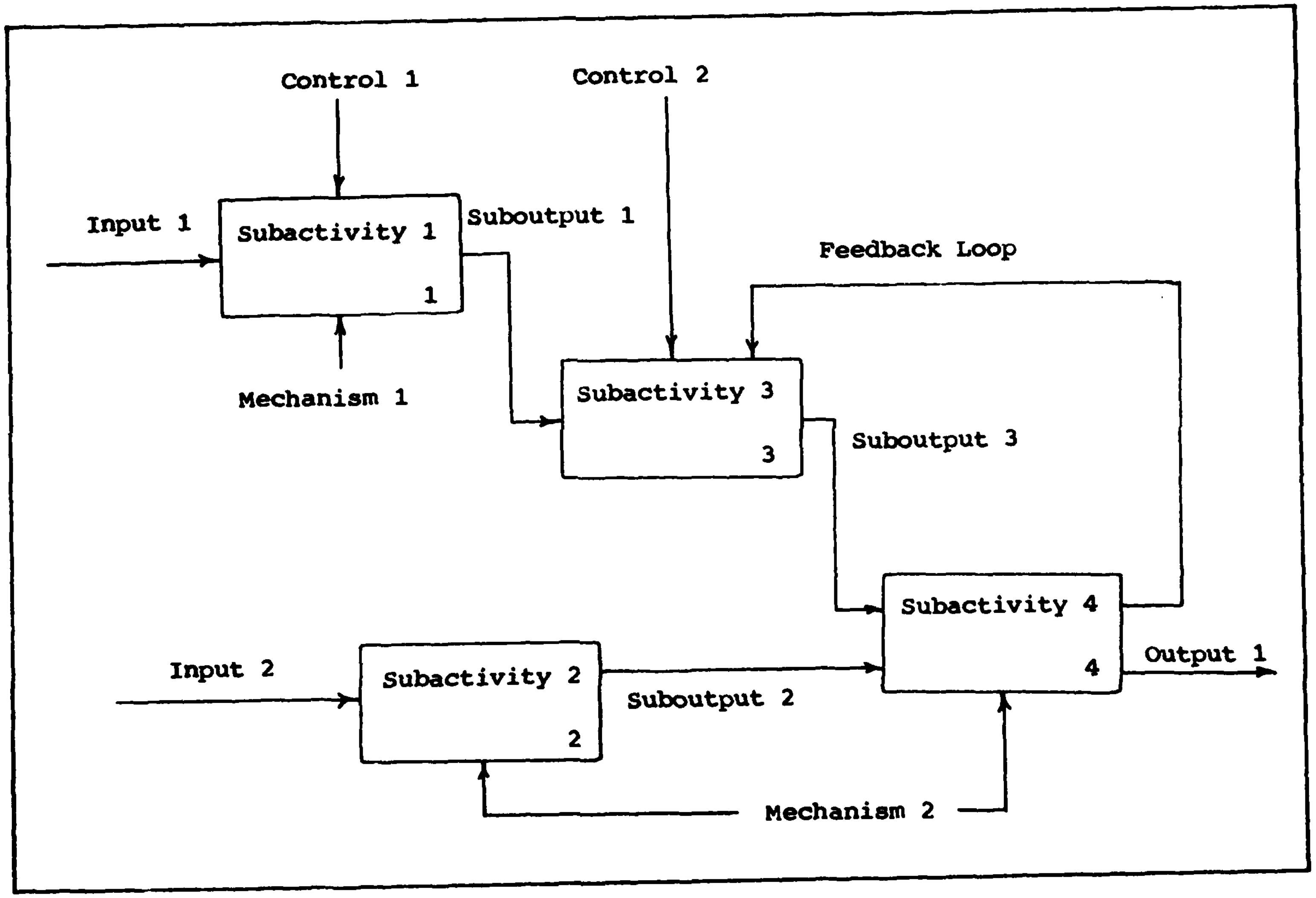

Figure 4. SADT Child Diagram.

entirely. All arrows on the parent diagram must be included in the child diagram. The boxes are numbered for easy reference.

Each subactivity is then decomposed into its major subactivities. This process is repeated until the system is described in sufficient detail.

Functional Analysis of the Bulk Data Network

SADT activity diagrams of the BDN are shown in Figures 5-10. The first diagram places the transmission activity in context. The second diagram outlines the overall tape transmission process and the last four diagrams detail the packetization, routing, packet transmission and reassembly processes.

As shown in the first diagram, Figure 5, the primary activity of the BDN is to transmit magnetic tapes, which are the only input and output of the system. The mechanisms used to transmit the tapes include: the input and output tape drives, the host computer at each site and the circuits which connect the computers. Each of these is a finite resource which must be carefully controlled. Controls are applied to the network at four levels:

1. The link protocols govern the transmission of packets between nodes.

2. The transport protocol or routing algorithm determines the path packets take to reach their destination.

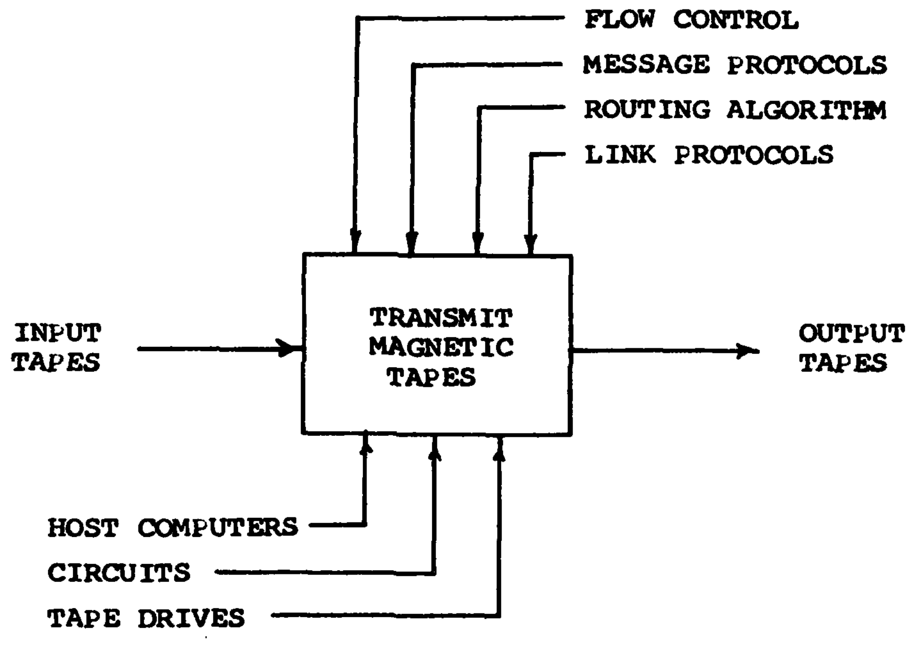

Figure 5. Bulk Data Network Operation.

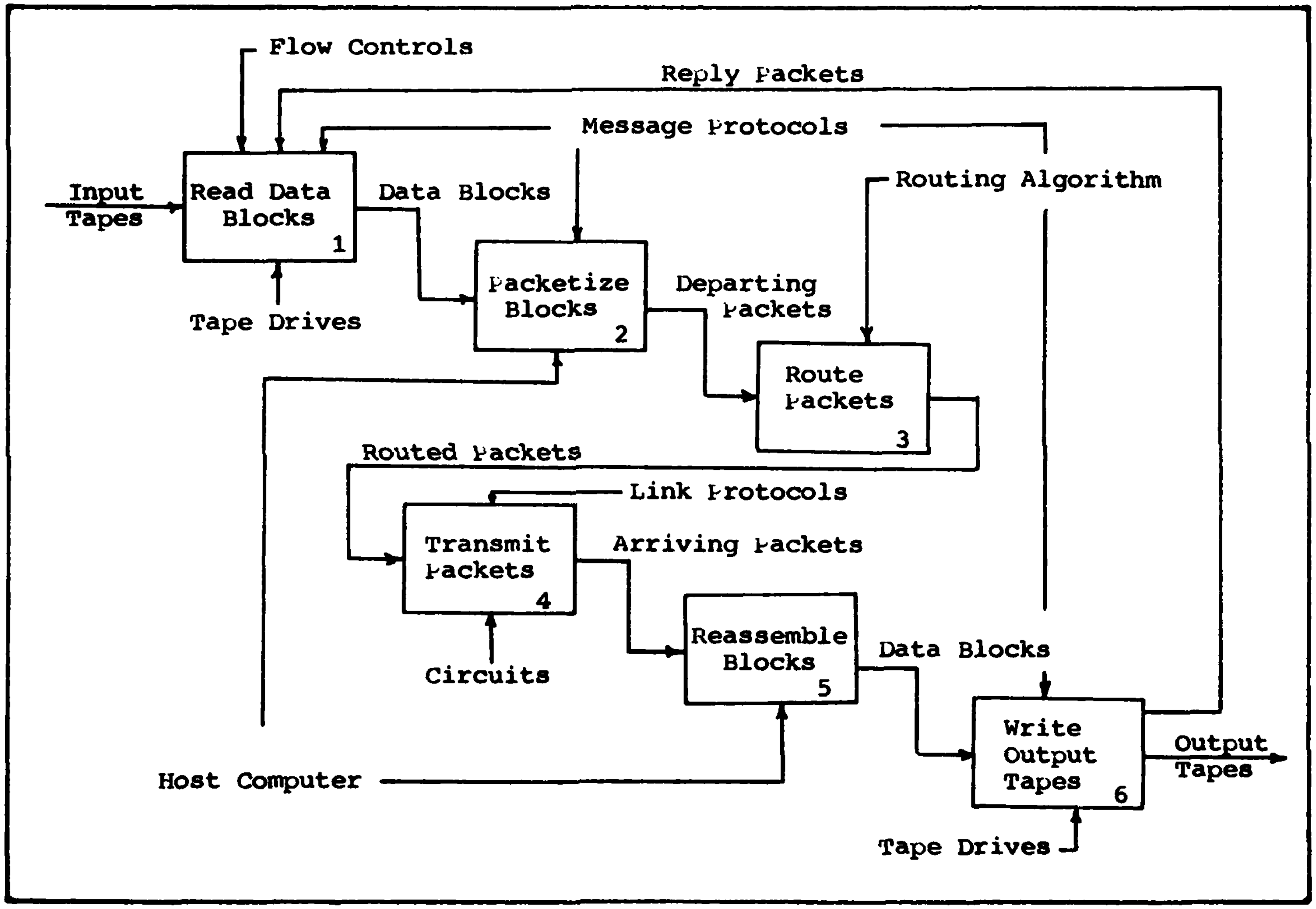

Figure 6. Tape Transmission Process.

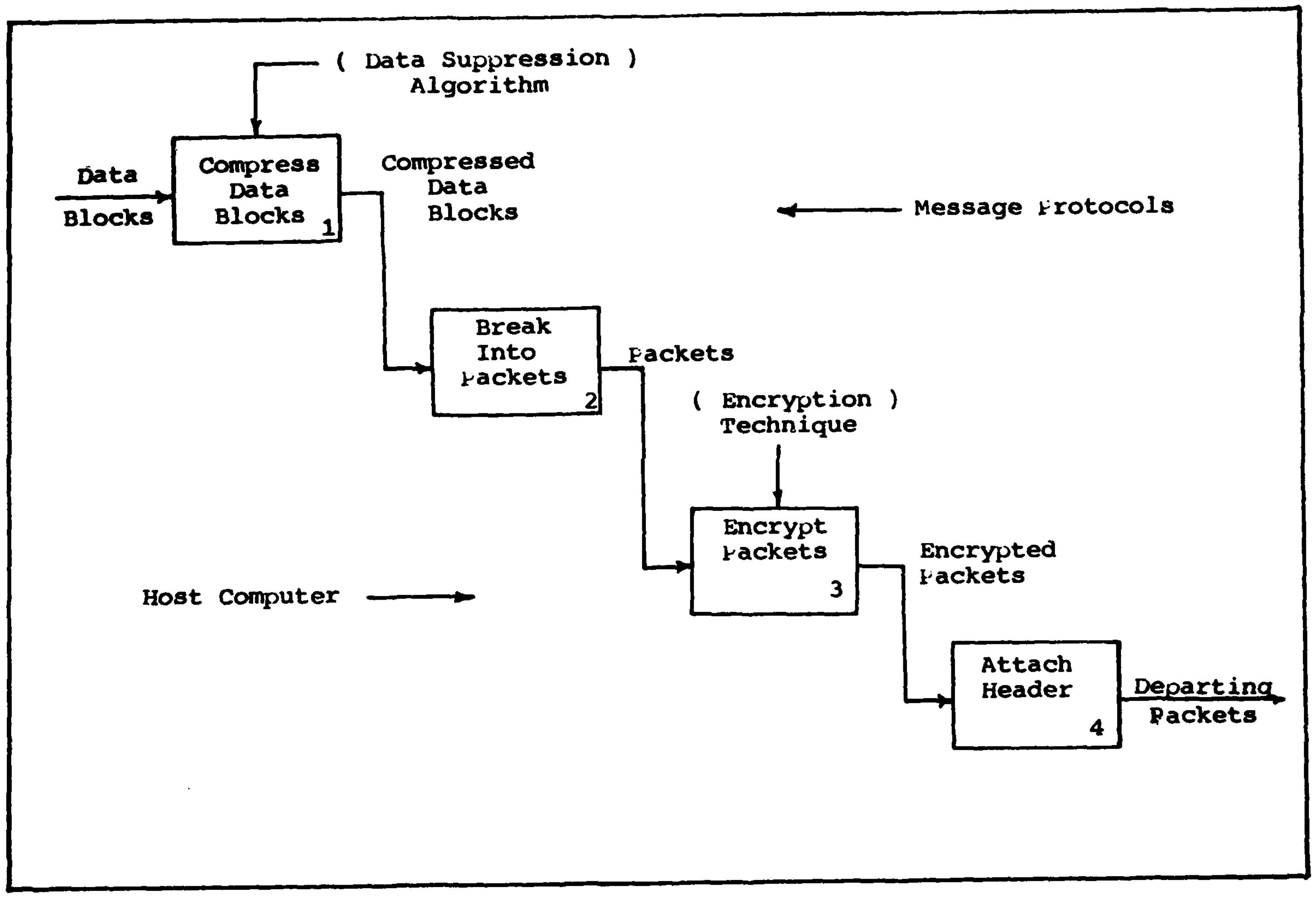

Figure 7. Packetization Process.

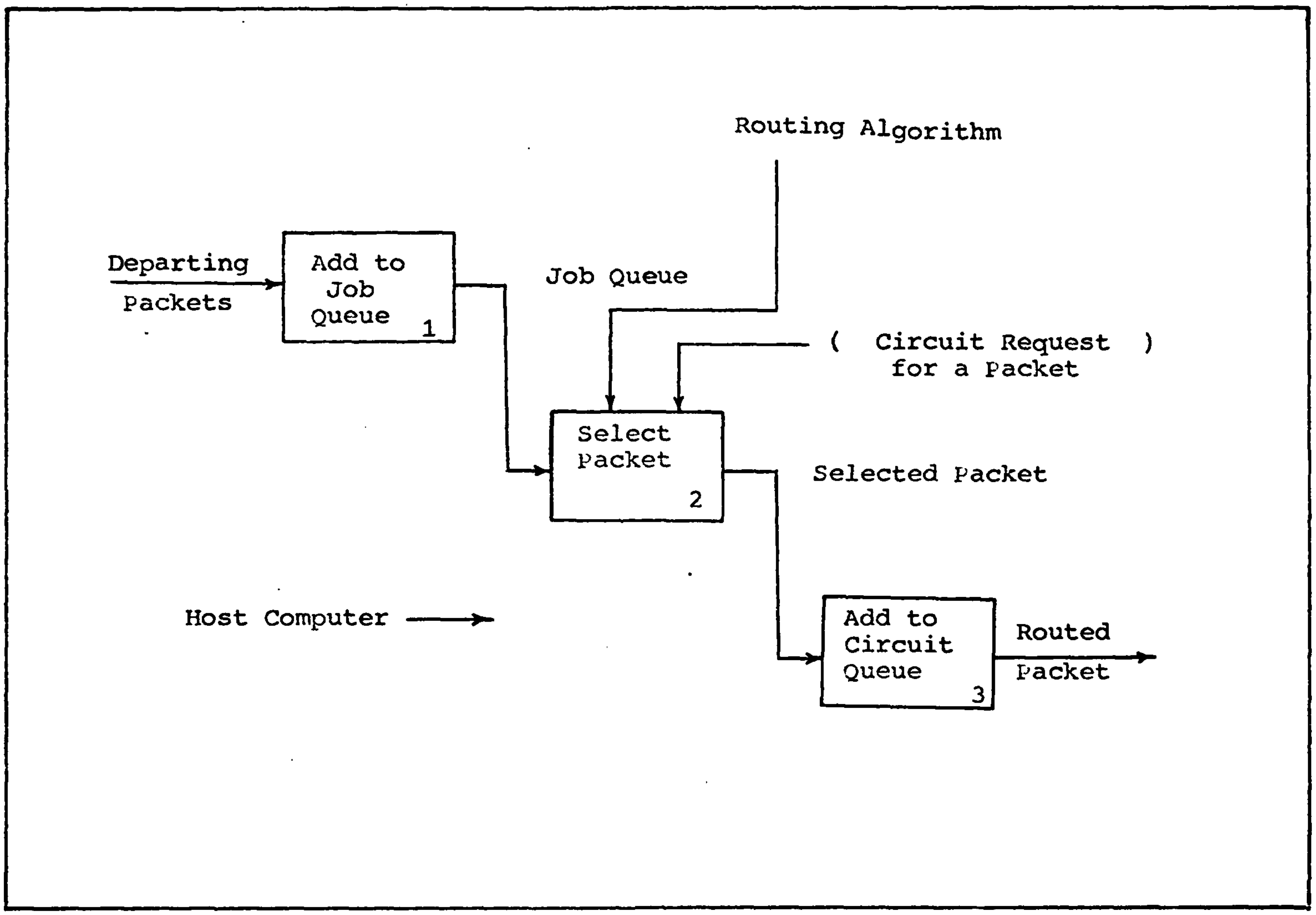

Figure 8. Routing Process.

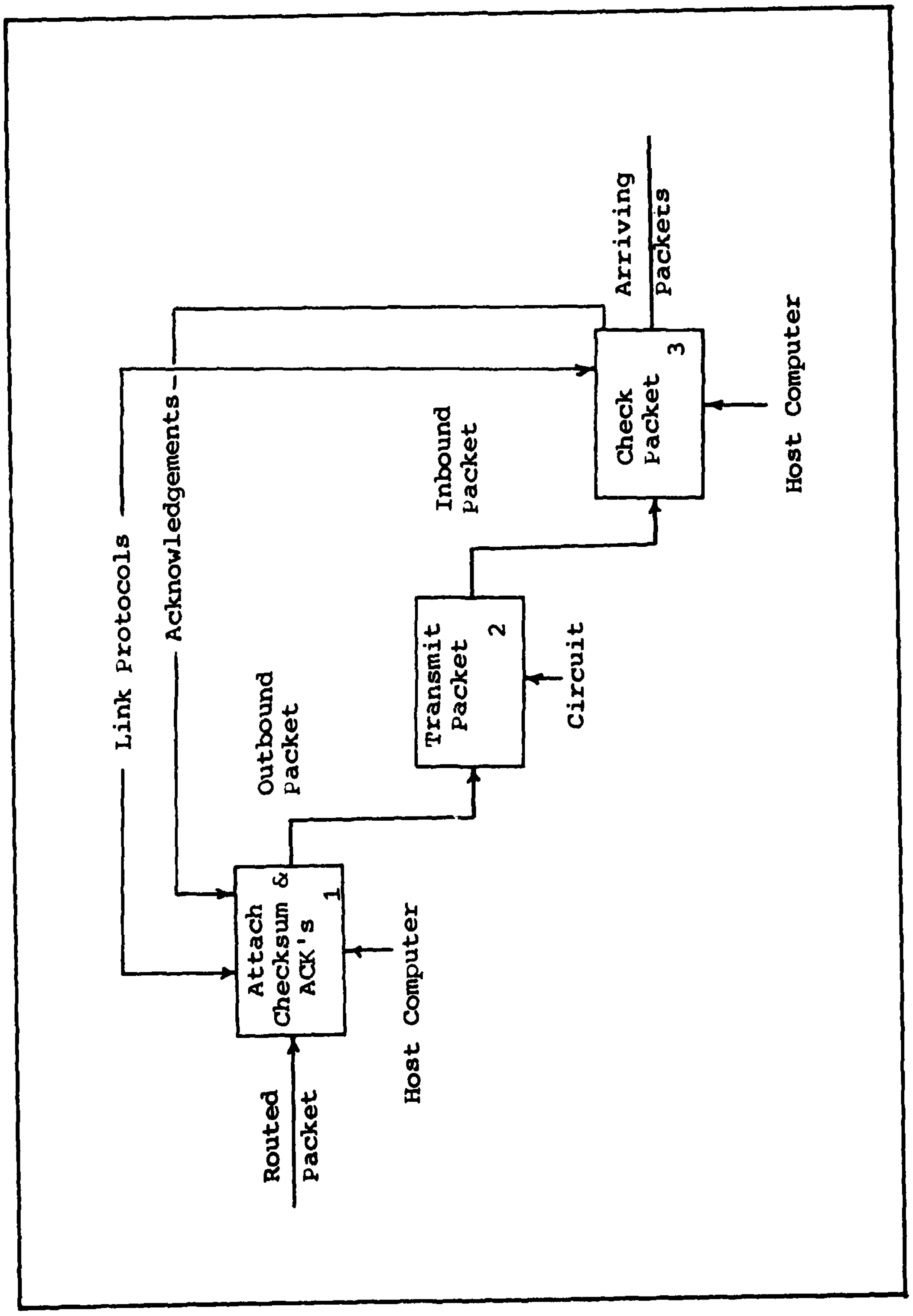

Figure 9. Packet Transmission Process.

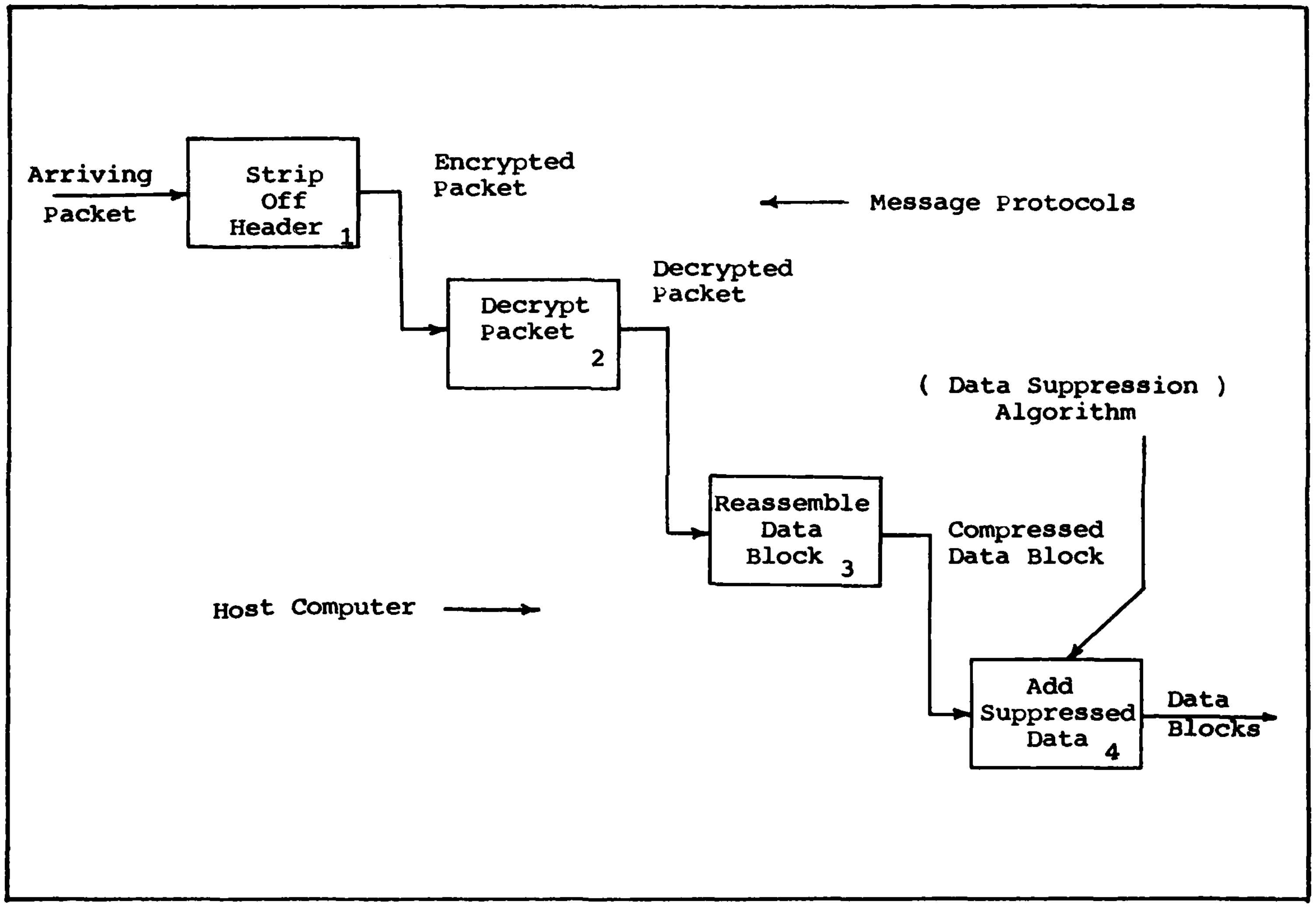

Figure 10. Reassembly Process.

3. The message protocols oversee the decomposition of blocks into packets at the source node and the reconstruction of packets back into blocks at the destination.

4. The flow control procedures regulate the input of packets to a given node to prevent overloading.

The tape transmission process consists of six major activities: block input, packetization, routing, packet transmission, reassembly and block output, (Figure 6). Tapes are read into the system at each host computer, one block at a time, using the five input tape drives. Next, the data blocks are processed and broken into packets by the host computer. The departing packets are routed toward their destination and placed on a circuit for transmission. If the next node is an intermediate node, the arriving packets are again routed toward their destination by placing them on a different circuit from the one they arrived on. The arriving packets are collected and reassembled by the host computer at the destination. Finally, the block is written to an output tape on one of the four output tape drives. When the block is safely on the output tape, a short 16 character reply packet is used to acknowledge receipt of the block.

Packetization. The packetization process is a four step process (Figure 7). The incoming data blocks are compressed using a data suppression algorithm which removes duplicate character strings. Next the compressed data is broken into data packets of 144 data characters each. The last packet in a block will contain between 1 and 144 data characters. The departing packets are encoded and a 30 character header is attached to the front of each packet. The packet header contains the packet's source, tape drive number, block number and a unique packet number used to reassemble the packets at the destination. The complete packet format is given in Appendix A.

Routing. In the routing process (Figure 8), the departing packets are linked into a general work queue. When a circuit is available, the job queue is searched to find the highest priority packet which can be output over the free circuit. The selection process is determined by the routing algorithm which will be discussed later. Finally, the selected packet is linked to the circuit's processing queue for transmission.

Packet Transmission. The packet transmission process has three subactivities (Figure 9). Prior to transmission, a mathematical computation is performed on the bits in the outgoing packet. The results of the computation, called checksums, are embedded in the packet's header. The packet is then transmitted. The incoming packet is checked for errors by repeating the checksum calculation and comparing the two sums. If they agree, the packet is good; if not, the packet contains an error and must be retransmitted.

The retransmission process is handled by the link protocols which are described in detail in the next section. The time required to transmit the packet is equal to the number of characters in the packet times the number of bits per character divided by the baud rate of the circuit. The transmission time for a 174 character data packets using a 4800 baud line is approximately a quarter of a second.

Reassembly. Arriving packets are collected at the destination node or a UNIVAC 1784 drum. When the last packet in the block arrives, the packets are retrieved from storage, the headers are removed, and the packets are decoded and reassembled into a block. Next, the suppressed data characters are added back and the block is written to the output tape. Finally, the block is acknowledged and the next block is read into the system.

Link Protocols

Occasionally, a packet will be lost during the transmission process. Approximately one in a hundred packets will be garbled in transmission. The link protocols are the mechanism used to detect transmission errors and retransmit a packet when errors are detected. The checksum procedure described above is used to detect errors.

Once an error has been found, a send and wait process is used to retransmit the packet. The process is demonstrated in Figure 11. Each site stores two numbers. The first is the "next channel sequence number" (NEXTCSN) and

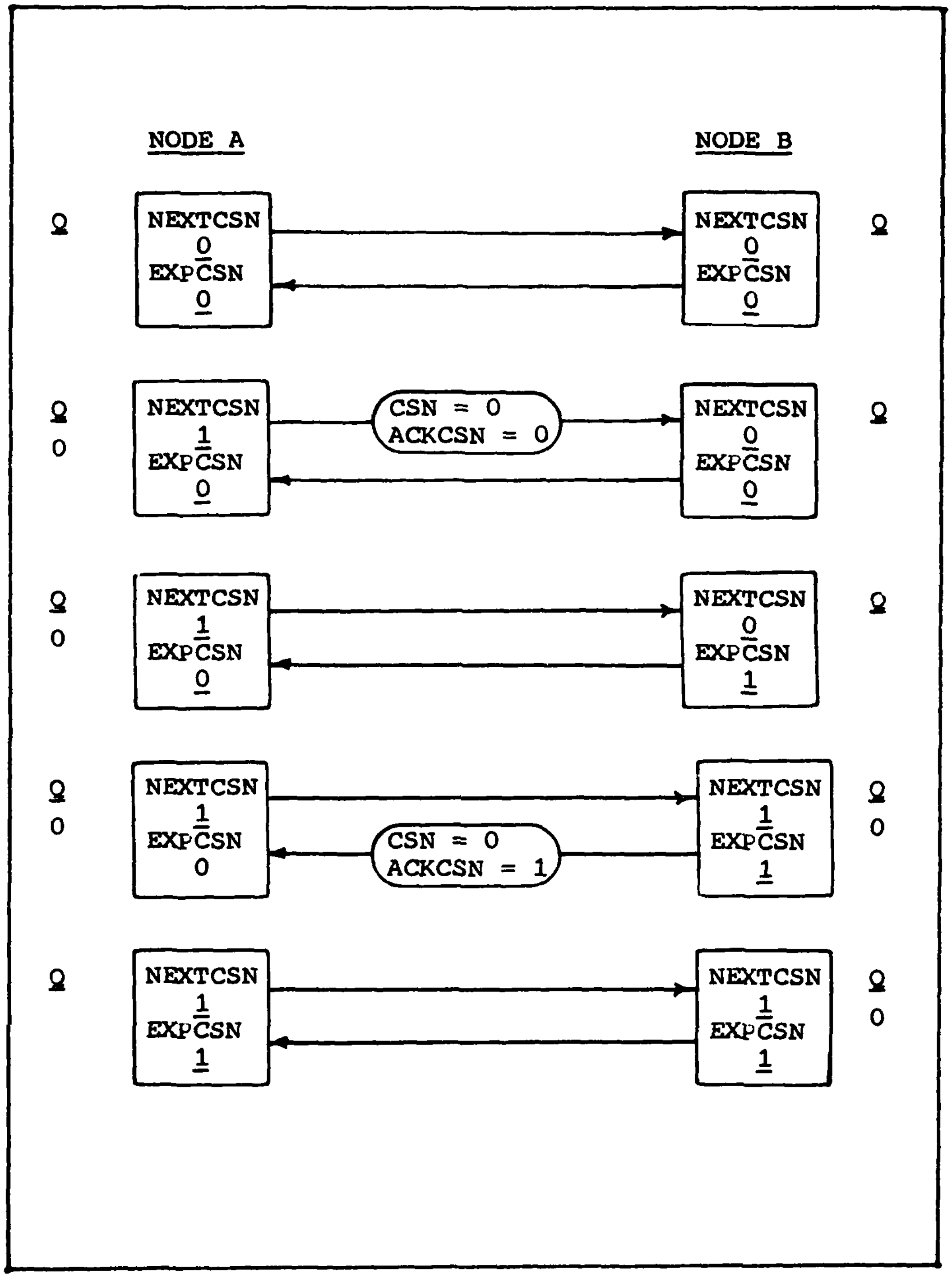

Figure 11. Link Protocols.

The second is the "expected channel sequence number" (EXPCSN). The NEXTCSN is the channel sequence number (CSN) that will be assigned to the next outgoing packet and the EXPCSN is the expected number of the next incoming packet.

At the beginning of the operation all numbers are set to zero. When the first packet is selected from the job queue at Node A, it is assigned CSN = 0, linked to a circuit queue and transmitted. Node A's NEXTCSN is incremented by 1, using Modulo 16 addition. When the packet arrives at Node B, it is checked for errors. If it is correct, the packets's CSN is compared with Node B's EXPCSN. If they match, the packet is accepted and Node B's EXPCSN is incremented; otherwise, the packet is out of sequence and is discarded. Node B acknowledges receipt of the valid packet by sending its EXPCSN back to Node A. It can do this by sending a short acknowledgement packet or, if it has another packet to send Node A, it can embed the acknowledgement (ACK) in the head of the outgoing packet. When Node A receives the ACK, it discards all the packets in its circuit queue with CSN's less than the ACKCSN. If Node A does not receive an acknowledgement in one second, it assumes that an error has occurred and retransmits the packet. Node A will retransmit the packet up to four times; if it cannot complete the transmission in four attempts, it assumes the circuit is down and returns the packet to the job queue for transmission on another circuit.

To reduce the time lost waiting for an acknowledgement, packets are transmitted in a four packet burst (i.e., Node A continues to transmit packets until it has four packets in the circuit queue). It then begins the one second wait. Node B will continue to acknowledge packets as they arrive, as long as it has packets going the other way. If it does not, Node B waits until it has received all four packets before generating a short acknowledgement packet. The circuit queue is emptied each time an acknowledgement is received, therefore, if there is traffic going in both directions and no errors occur, neither side will lose time waiting for a reply. However, if an error occurs, a full four packet burst has to be retransmitted.

Schwartz (13) suggests a simple improvement to this technique which reduces the wait time to detect an error. His proposal calls for a short negative acknowledgement (NAK) to be sent as soon as the error is detected. The proposal is discussed further in the next chapter.

Routing Algorithm

The routing algorithm is a set of policies and routing tables that determine the path an individual packet will follow to reach its destination. A packet is routed out of a node on the first available circuit which offers a non-looping path to the packet's destination, with the following exceptions:

1. A packet will not be output on the circuit it arrived on.
2. A packet will not be sent back to its source node.
3. If the current node is adjacent to the packet's destination, the packet must be output over the connecting circuit.

The exceptions are necessary to ensure that packets do not loop. To determine if a packet can be output on the free circuit, each node maintains a list of nodes that can be reached via a given circuit. The lists are stored in a routing table (Figure 12). A "1" in the table indicates that the node can be reached using the circuit. A "0" indicates that it cannot. The routing table is constructed based on the node's knowledge of the current network connectivity.

Each node maintains a knowledge of the network connectivity on a per node basis. When a given node reloads, it requests connectivity information from each neighboring node with which it establishes a direct connection. The neighboring nodes respond by transmitting a short (16 character) status packet that indicates which circuits are operating. Each circuit is identified by a specific bit in the status packet. The bit is set to "1", if the circuit is up. Subsequent changes to the connectivity knowledge of the node are made in an asynchronous manner. Each node

<u>Routing Table for Node 3</u>

Node \ Output Port Number	1	2	3
1	1	0	0
2	1	1	1
3	0	0	0
4	1	1	1
5	0	0	1
6	0	1	0

Figure 12. Sample Routing Table.

transmits a status packet to each of its neighbors every twelve seconds. The times are staggered so that one node is transmitting every two seconds. In this way, every node will be aware of a change in connectivity within 30 seconds.

The connectivity information is stored in an adjacency matrix. The adjacency matrix is updated by matching the incoming status packet with a circuit table, which identifies the source and destination of each circuit. The process is demonstrated in Figure 13.

Circuit Table

Number	Node	to Node
1	3	6
2	3	5
3	4	5
4	4	6
5	2	4
6	1	3
7	1	2

Status Packet

Status
1
1
1
1
1
1
1

Adjacency Matrix

	Destinations					
Source	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	0	1	1
4	0	1	0	0	1	1
5	0	0	1	1	0	0
6	0	0	1	1	0	0

Figure 13. Table Update Process.

The adjacency matrix and the algorithm outlined below are used to construct the routing table. The algorithm makes use of two abstract data types: A Stack and a Nodes_Visited set. A Stack is a storage structure used to hold node numbers. Four operations are defined on the Stack:

1. INITIALIZE (STACK) which initializes the stack pointer to nil.
2. PUSH (NODE) which adds a node number to the top of the stack.
3. POP (NODE) which returns the last node number placed on the stack.
4. STACK_EMPTY which returns true if the stack is empty.

The Nodes_Visited set is a set which contains the node numbers. Four operations are also defined on this structure:

1. INITIALIZE (NODES_VISITED) which sets Nodes_Visited to the null set.
2. VISIT (NODE) which adds the node number, "node," to the set.
3. VISITED (NODE) which returns true if node number "node" is in the set.
4. ALL_NODES_VISITED which returns true if the set is complete.

The function shown in Figure 14 returns a list of nodes that can be reached by a non-looping path for each output circuit. Each list, with the destination of other output circuits removed, becomes a column in the routing table.

When a circuit is ready for another packet, the host computer searches the job queue for the first packet that is not already linked to a circuit queue, has destinations that can be reached using the free circuit and does not violate the three exceptions previously stated. Since both the packet's destination and the circuit's entry in the routing table are single words, the computer can easily check to see if the sets have a non null intersection in a single operation. The selected packet is linked to the circuit queue and its down line routing is set equal to the intersection of the two sets. After transmission, these nodes are removed from the packet's destination. If no more destinations remain, the packet is discarded; otherwise, the packet is made available for output on another circuit.

Message Protocols

Since the individual packets within a block are free to travel different routes, there is no guarantee that the packets will arrive in order. Therefore, tape to tape or message protocols are necessary to control the transmission process at the sending node and the collection process at the receiving node. The message protocols are similar to the link protocols.

```
FUNCTION FIND_REACHABLE_NODES
   FOR EACH OUTGOING CIRCUIT DO
      INITIALIZE (STACK)
      INITIALIZE (NODES_VISITED)
      VISIT (CURRENT_NODE)
      PUSH (CIRCUIT_DESTINATION)
      VISIT (CIRCUIT_DESTINATION)
      REPEAT
         POP (NODE)
         FOR DESTINATION = 1 TO LAST_
                                        DESTINATION DO
            IF ADJACENCY_MATRIX (NODE, DESTINATION)
                                   = 1 AND NOT VISITED
                                   (DESTINATION) THEN
               PUSH (DESTINATION)
               VISIT (DESTINATION)
            END_IF
          END_FOR
       UNTIL STACK_EMPTY OR ALL_NODES_VISITED
       REACHABLE NODES FOR CIRCUIT = NODES_VISITED -
                                            CURRENT_NODE
   END_FOR
END_FIND_REACHABLE_NODES
```

Figure 14. Algorithm 1, Find_Reachable_Nodes.

Each packet is assigned a block control number (BCN) and a packet control number (PCN) when the block is broken into packets. The last packet in the block is marked to indicate that it is actually the last packet in the message. When the last packet has been placed in the job queue, a 30 second timer is started and a "send control table" (SCT) is set up for the block. The SCT is a set which contains a list of the tape's destinations. As replies are received from the distant end, the locations are deleted from the table. When all destinations have responded, a new data block is read into the system. If the 30 second timer goes off before the block is fully acknowledged, the block is retransmitted to the unacknowledged sites.

To keep the circuits busy, the sending node will begin processing the second data block before the first block has been acknowledged. However, it will not process the third block until the first block is fully acknowledged. In this way, a tape can have no more than two blocks in transit at the same time. Note, if the 30 second timer runs out, both the first and second blocks are retransmitted.

When the first packet in a block arrives at a destination, the receiving node sets up a "receiving control table" (RCT) for the block. As subsequent packets arrive, the appropriate bit is set to "1". Duplicate packets are discarded. When the last packet in the block arrives, the table is examined to see if all the packets have arrived. If so, a reply packet is sent to acknowledge receipt of the block. Otherwise, a two second timer is started. If the remaining packets do not arrive in two seconds, the destination node sends a reply packet requesting retransmission of the missing packets.

<u>Flow Control</u>

Flow control is the process which keeps the network from overloading by controlling the inputs to a node. Packets come from three sources: the input circuits, the tape drives and within the node itself (e.g., control packets). A node can only store 90 packets in its job queue at a time.

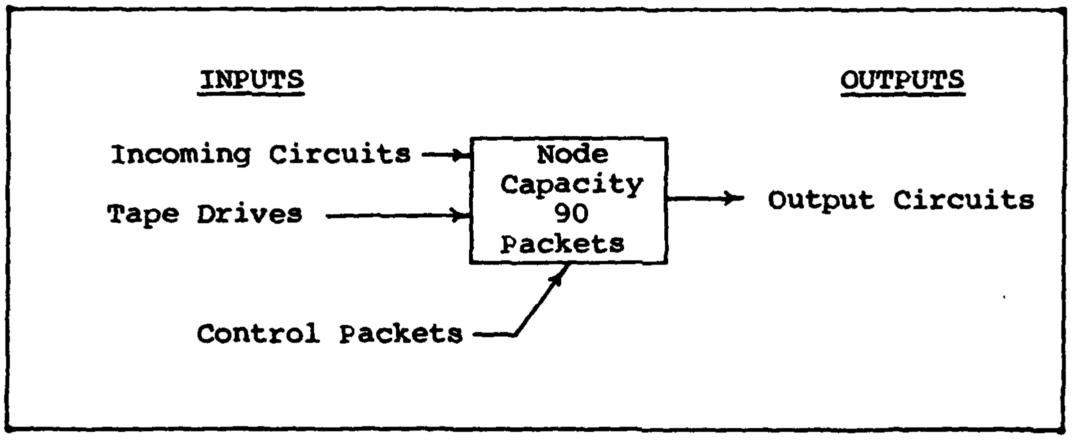

Figure 15. Flow Control.

If the queue is full, incoming packets will be denied entry to the node. This can result in a deadlock if two adjacent nodes with full buffers are trying to transmit to each other. Neither node can accept the incoming packet nor will they be able to discard the outgoing packet. To prevent this from happening, the following rules are applied.

1. Control packets are given priority over data packets.

2. If the job queue is more than 70% full, no more data packets will be allowed to enter the node from the tape drives.

3. If the job queue is full, the adjacent nodes are told to send only one packet at a time.

These actions slow down the inputs to the node and allow the output circuits to empty the buffer.

Summary

The tape transmission activity is a six step process consisting of block input, packetization, packet routing, transmission, reassembly and block output. Tape blocks are read into the host computer at the source node using one of five input tape drives. The blocks are then compressed, packetized and encoded. Next, the packets are routed and transmitted over 4800 baud circuits to the destination. At the destination, the packets are decoded and reassembled into blocks by the host computer. Finally, the blocks are written out by one of the four output tape drives and an acknowledgement is sent back to the source.

Controls are applied at four levels. Link protocols detect and correct transmission errors. A routing algorithm determines packet routing. Message protocols govern the packetization and reassembly processes and flow control procedures limit the amount of traffic on the network at any one time to prevent deadlocks.

The next chapter describes what changes must be made to the above procedures in order to add the additional circuit and interactive terminal traffic.

IV. Proposed Changes to the Network

The last chapter described the network as it exists today. This chapter describes several proposed changes to the network, which are to be evaluated using the simulation model. These include proposals to increase line capacity, change the routing algorithm, modify the link protocols to incorporate negative acknowledgements, and add interactive terminal traffic.

Increased Line Capacity

AFCCPC/SKDAR will add a new 9600 baud full duplex circuit from OO-ALC to HQ AFLC and replace the SM-ALC to OO-ALC circuit with a new 9600 baud circuit. These additions will increase the network's performance by increasing its physical capacity. Since the circuit's capacity is not a factor in packet routing, SKDAR does not plan to make significant changes to the network software. The routing tables at OO-ALC and HQ AFLC will simply be enlarged to include the additional circuit.

Alternative Routing Algorithms

It can be shown by an exhaustive search of all possible routings that the current BDN routing algorithm prevents packets from looping in the present configuration; however, this is not the case in general. For example, in the upgraded network, if the SA-ALC to WR-ALC circuit is temporarily

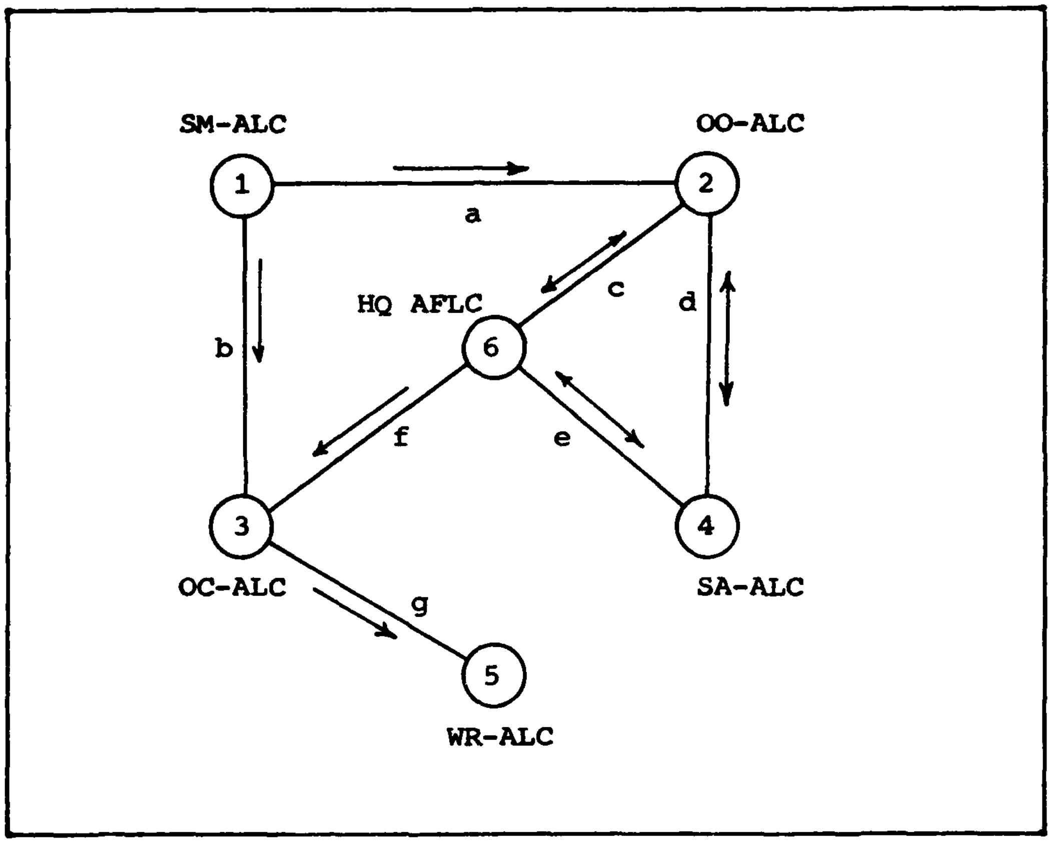

Figure 16. Looping Path Example.

down (Figure 16), a packet originating at Node 1 destined for Node 5 can be output on either Circuit a or b. If a is chosen, the packet goes to Node 2; from here it can go out Circuits c or d. If c is chosen, the packet goes to Node 6. Now the packet can go out Circuits e or f. If e is used, the packet goes to Node 4. From Node 4, the packet must go back to Node 2, forming a loop.

Kuo (6) and Schwartz (13) describe several alternative routing strategies which prevent looping no matter what the connectivity of the network. Two of these, minimum hops (MINHOP) and minimum plus one hops (MIN+1HOP), are basically extensions of the BDN routing method and could be implemented without extensive modification to the current system. In the MINHOP strategy, each packet takes the path which requires the fewest number of transmissions between nodes (hops). In the MIN+1HOP algorithm, each packet is free to take either a minimum hop route or a minimum plus one hop route. Packets cannot loop in either case, because a looping path would always require at least two more hops than a non-looping path using the same nodes. This is demonstrated using the directed graph in Figure 17.

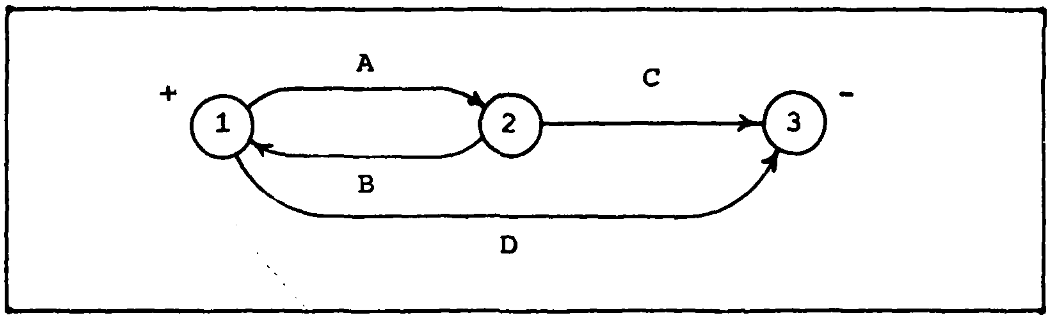

Figure 17. Directed Graph.

There are an infinite number of paths from Node 1 to Node 3; they can be represented by the regular expression

$$(AB)^N AC + (AB)^N D \qquad (1)$$

where N is integer ≥ 0. The expression is read path AB taken N times followed by path AC or path AB taken N times followed by path D.

The term $(AB)^n$ represents the loop from Node 1 to Node 2 which can be taken any number of times before moving on to Node 3 using either path D or path AC. A looping path between two adjacent nodes of the form $(ab)^N$, will contain 2N hops. A looping path between three nodes, $(abc)^N$, will contain 3N hops. In general, a looping path between m nodes, $(abc...m)^N$, will contain mN hops. If a node is not permitted to transmit to itself, then the minimum length a looping path can have is two hops.

By definition, a looping path always returns to the node it started, therefore, we can eliminate all the looping terms in Equation 1 and still be certain of reaching Node 3. This leaves:

$$AC + D \qquad (2)$$

as the only non-looping paths. In this example, path D is the minimum hop path with a length of one. Since, the minimum length of a looping path is two, if a loop is added to the minimum path, the total length would have to be three or greater. Therefore, we can select any path of length 1 or 2 and be certain that it will not loop. This is a sufficient but not a necessary requirement, since in a larger network, there can exist non-looping paths with path lengths greater than the minimum hop length plus one.

The minimum hops method can be implemented by simply substituting a MINHOP table for the Reachable_Nodes table and modifying the Find_Reachable_Nodes algorithm to find minimum paths instead of non-looping paths. The selection of packets from the job queue will remain the same. We no longer need to worry about selection of a direct path to an adjacent node, since this route will always be the minimum hop path.

The MINHOP table is made from a hops table, which is made using a modified version of the Find_Reachable_Nodes algorithm. The Find_Minimum_Hops algorithm, shown in Figure 18, uses two stacks, STACK (0) and STACK (1). Every time the search step changes rows, the algorithm switches stacks and the depth of the search is increased by one. When a node is visited, the current value of depth is placed in the node's entry in the HOPS table. All hop table entries are set to 99 (infinite) before the search begins. Nodes which cannot be reached by a non-looping path will still be set to infinite when the algorithm terminates.

The MINHOP table is formed by the process shown in Figure 19. First, the minimum hop path for each destination is identified. Next, the MINHOP table entries are determined by comparing the HOPS table entry to the minimum path value.

```
FUNCTION FIND_MINIMUM_HOPS
   INITIALIZE HOPS_TABLE ENTRIES TO 99 (INFINITE)
   FOR EACH OUTPUT CIRCUIT DO
      DEPTH = 1
      VISIT (CURRENT_NODE)
      PUSH (CIRCUIT_DESTINATION) ON STACK (0)
      VISIT (CIRCUIT_DESTINATION)
      HOPS_TABLE (OUTPUT_CIRCUIT, CIRCUIT_
                                 DESTINATION) = DEPTH
      DEPTH = DEPTH + 1
      REPEAT
         POP (NODE) FROM STACK (DEPTH MOD 2)
         FOR DESTINATION = 1 TO LAST_DESTINATION DO
            IF (ADJACENCY [NODE, DESTINATION] = 1)
                                       AND NOT VISITED
                                    (DESTINATION) THEN
               PUSH (DESTINATION) ON STACK (DEPTH +
                                            1 MOD 2)
               HOPS_TABLE (OUTPUT_CIRCUIT,
                                 DESTINATION) = DEPTH
               VISIT (DESTINATION)
            END_IF
         END_FOR
         IF STACK (DEPTH MOD 2) EMPTY THEN
            DEPTH = DEPTH + 1
         END_IF
      UNTIL STACK (DEPTH MOD 2) EMPTY OR ALL_NODES_
                                             VISITED
   END_FOR
END_FIND_MINIMUM_HOPS
```

Figure 18. Algorithm 2, Find_Minimum_Hops.

Sample MINHOP Calculations for Node 3
(Upgraded Network)

HOPS TABLE

	Circuit		
Node	1	2	3
1	1	3	4
2	2	2	3
3	0	0	0
4	3	2	2
5	4	3	1
6	3	1	3

Minimum Hops

Minimum Hops
1
2
0
2
1
1

MINHOP TABLE

	Circuit		
Node	1	2	3
1	1	0	0
2	1	1	0
3	0	0	0
4	0	1	1
5	0	0	1
6	0	1	0

Figure 19. Calculations for MINHOP Table.

Implementation of the minimum plus one hops algorithm is similar to the minimum hops method. Each node stores both a MINHOP table and a MIN+1HOP table. The MIN+1HOP table contains a list of the nodes that can be reached using either a minimum hops route or a minimum plus one hop route. The MIN+1HOP table is used in the packet selection process, if the packet originated at the current node, otherwise, the MINHOP table is used. Since routing tables are changed in an asynchronous manner, it is possible for a selected packet to have arrived on the requesting circuit or to have originated at the circuit's destination. When this happens, the packet is discarded from the job queue and another packet is selected.

The adjacency matrix for a network with n nodes contains n^2 entries. Since both the Find_Reachable_Nodes and Find_Min_Hop algorithms may have to search each entry in the table, the search time for each algorithm is on the order of n^2, written $O(n^2)$. If the network is lightly connected, most of the entries will contain a zero. Therefore, a large amount of time is wasted checking unnecessary entries.

This time can be saved if the adjacency matrix is replaced by a connectivity table. The connectivity table can be represented by an array of linked lists, one list for each node, or by a two dimensional array. In either implementation, the connectivity table contains a list of the nodes which are adjacent to each node. If a set of linked lists is used, each item in the list contains a node number and a pointer to the next item in the list. The last item in a list points to a nil location. The total number of items in the connectivity table is equal to two times the total number of circuits.

The need for pointers can be eliminated, if a matrix structure is used. The size of the array can be limited to n x m, where n is the number of nodes and m is the maximum number of output circuits per node. A "0" entry is used to indicate the circuit is down or the output port is not used. When the connectivity matrix is used, the search time is O(mn). In a lightly connected network, m is substantially less than n and the improved algorithm is much faster than the original. If the network is highly connected, the search time is essentially the same.

Negative Acknowledgements

The BDN uses four bits to store the CSN's in each packet. Schwartz (13) has shown that three bit Modulo 8 CSN's are sufficient when the maximum burst size is four packets. This fourth bit can be used to designate the packet as a negative acknowledgement. The NAK would be used to signal an error as soon as it is detected by the receiving node. The NAK would be embedded in a returning packet, if one exists, or a short NAK packet would be sent.

Upon receipt of the NAK, the transmitting node would discard all the packets in its circuit queue up to but not including the one with the NAK CSN. It would then retransmit the burst beginning with this packet. By acknowledging errors as soon as they are detected, the lost transmissions time can be reduced from one second plus four transmissions to approximately two packet transmissions; one to detect the error and one to retransmit the packet. If the NAK is lost, the old time-out process would catch the error.

Interactive Terminals

During the course of this study, HQ AFLC/LO asked SKDAR to determine if the BDN could support ten terminals (five at WR-ALC and five at SM-ALC) connected to a computer at HQ AFLC with a maximum round trip response time of two seconds. We were asked to include terminals in the model. The terminals are expected to generate 1,000 packets each per day. They will be processed the same as the tape data packets, but will be given a higher priority. Terminal packets will not be allowed to interrupt a tape data packet or be acknowledged from the distant end.

Unlike the tapes, in which new packets are deterministically generated based on replies from the destination, the time between terminal packet arrivals is a random process. The arrival of packets at a source node is described by the Poisson distribution (13). If the average arrival rate is

λ units per hour, the interarrival time distribution is an exponential with a frequency function f(t) given by

$$f(t) = \lambda e^{-\lambda t}, t \geq 0 \tag{3}$$

where t is the time between arrivals.

To simulate the process, a method is needed to generate random interarrivals times that satisfy Equation 3. Meirer (7) gives such a method. First, we find the cumulative distribution function F(t) by integration.

$$\begin{aligned} F(t) &= \int \lambda e^{-\lambda t} dt \\ &= -e^{-\lambda t} + C \end{aligned} \tag{4}$$

Making use of the fact that at t = 0, F(t) = 0, we substitute to find the value of C.

$$0 = -e^{-0} + C \tag{5}$$

and

$$C = e^{-0} = 1 \tag{6}$$

Therefore, the cumulative distribution function is

$$F(t) = 1 - e^{-\lambda t} \tag{7}$$

λ

Since F(t) varies between 0 and 1.0 based on the value of t, we can substitute a random variable X, which varies between 0 and 1.0 for F(t), to obtain an expression for t.

$$X = 1 - e^{-\lambda t} \qquad (8)$$

$$e^{-\lambda t} = 1 - X \qquad (9)$$

Taking the natural logorithm of both sides

$$-\lambda t = \ln(1 - X) \qquad (10)$$

$$t = -1/\lambda \cdot \ln(1 - X) \qquad (11)$$

Equation 11 can be used to find exponentially distributed random interarrival times by substituting random numbers, between 0.0 and 1.0 for X and solving for t. Since X is uniformly distributed between 0 and 1, so is (1 - X), therefore, Equation 11 can be simplified by substituting X for (1 - X) to obtain

$$t = -1/\lambda \cdot \ln(X) \qquad (12)$$

Equation 12 can produce both very long and very short interarrival times, which do not actually occur in the real system. Therefore, to preclude these times from occurring, a maximum and a minimum interarrival time are specified; if the generated value of t exceeds either value, the appropriate boundary value is substituted.

Summary

This chapter has shown that SKDAR's proposal to increase the number of circuits from seven to eight will make the network vulnerable to packet looping. This can result in reduced packet throughput. Two alternative routing algorithms, MINHOP and MIN+1HOP, were proposed to remedy this situation. These methods prevent looping by restricting the routing selection to either a minimum hop or minimum plus one hop path.

The link protocols can be made more responsive to transmission errors if a negative acknowledgement is used to signal an error. NAK's can be implemented by reducing the CSN's from four bits to three bits and using the extra bit to designate the ACK as a NAK.

At the end of the chapter, a formula was developed for generating random interactive packet arrivals; it can be used in the model to study the terminal proposal. The next chapter uses this information to develop the computer simulation.

V. Program Design

The functional description presented in the last two chapters was used to develop a computer simulation of the network. A single program was designed to simulate both tape and terminal packet transmissions using any of the three routing algorithms described in the last chapter. The model makes extensive use of abstract data structures to represent basic network components. The components are linked together to form a network. The basic objective was to provide a flexible model, where the user can specify the network's connectivity and architecture at run time. This allows him to model different network configurations without changing the program.

Network Model

The program is a model network in which packets are generated at source locations, moved through the network using circuits, and then collected at a destination node. It consists of a set of data structures, which serve as inputs, outputs and mechanisms, and a set of procedures, which performs the packet transmission and network control functions. Four primary data structures or modules are used to construct the model (Figure 20). These include: nodes (host computers), circuits, tape drives and terminals. Each site can have one host computer, up to four outgoing

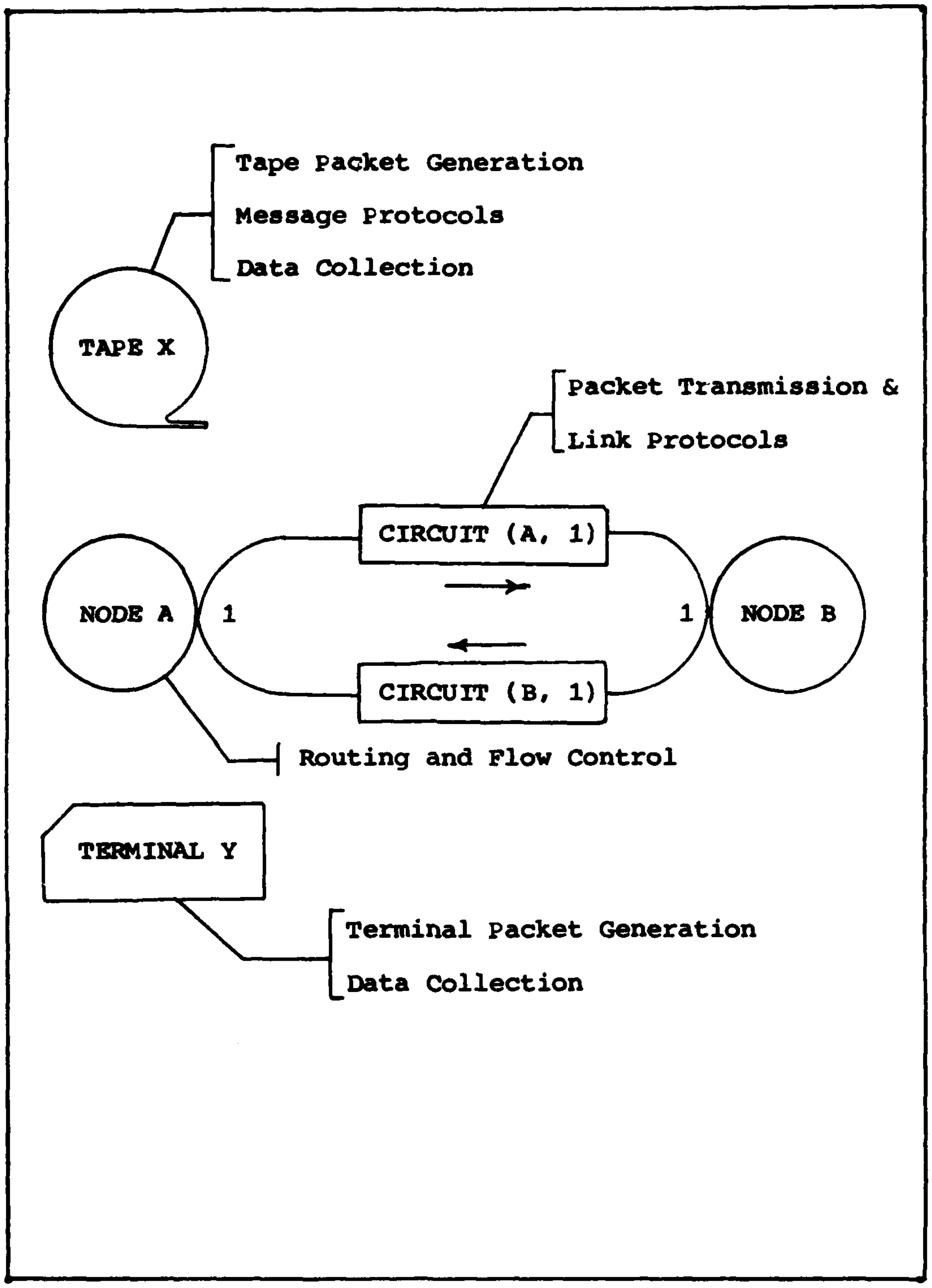

Figure 20. Network Components.

and four incoming circuits, up to five tape drives and any number of terminals. The actual network configuration is specified by the user through input variables. Circuits are identified by a node number and a port number. Nodes, tape drives and terminals are numbered sequentially from one to the total number of units in the network.

Specific network activities can be identified with each device. The tape drives and terminals provide input to the network and perform the message control and data collection activities. Nodes perform the routing and flow control activities and circuits provide packet transmission and error control. Each circuit provides one way packet transmission from the circuit's source node to a destination node; therefore, two circuits, a primary and a return, are required for full duplex transmission.

Each activity has a finite duration. At the beginning of the activity, network resources and raw materials (packets) are committed to the process and remain committed until the activity is completed. When the activity terminates, the resources are released by the activity for use by other processes and the outputs are moved along to the next operation. An event is the boundary between two activities and is assumed to have no duration. If there is a delay between the end of one activity and the beginning of the next, a dummy delay activity is created to separate the two activities. The time at which an event occurs,

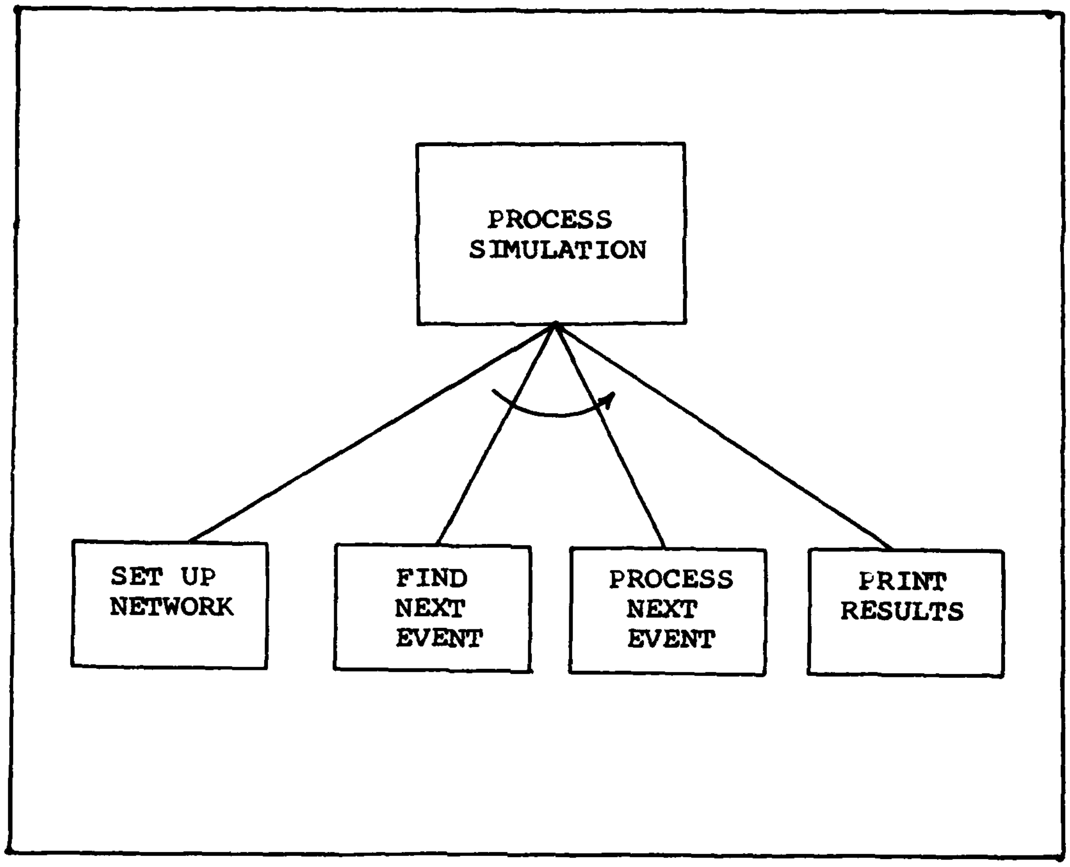

Figure 21. Simulation Process.

called the event time, is equal to the time of the last occurrence of the event plus the duration of the interceding activity.

The simulation is a four step process (Figure 21). It begins by reading user-supplied input data, setting up data structures and calculating an initial event time for each event. Next, the program finds the first event to be processed by searching for the event with the earliest time. The simulation clock is set equal to this time and the name of the event is passed to the process event function for processing. The event is processed by changing variables to reflect the commitment of resources and the consumption of raw materials, or the freeing of resources and the production of finished goods. The Find_Next_Event/ Process_Next_Event operations are repeated until a pre-defined terminating condition is met. This condition is usually a maximum run time or the production of a specified number of outputs. When the loop terminates, the results are printed and the program stops.

Network Events

There are ten events used in the simulation. Eight are network events: Tape_Stop_Send, Tape_Stop_Receive, Packet_Ready, Status_Update, Begin_Transmission, End_ Transmission, Reset_Error and New_Arrival; and two are overhead events: Stop and Collect_Job_Queue_Lengths. The overhead events are non-network events used to collect data and stop the simulation. Each network event is associated with a specific device (Figure 22). Tape_Stop_ Send and Tape_Stop_Receive are tape drive related events. Packet_Ready and Status_Update are node events. Begin_ Transmission, End_Transmission and Reset_Error are circuit events and New_Arrival is a terminal event. Each event signals the program that a specific activity is ready to begin.

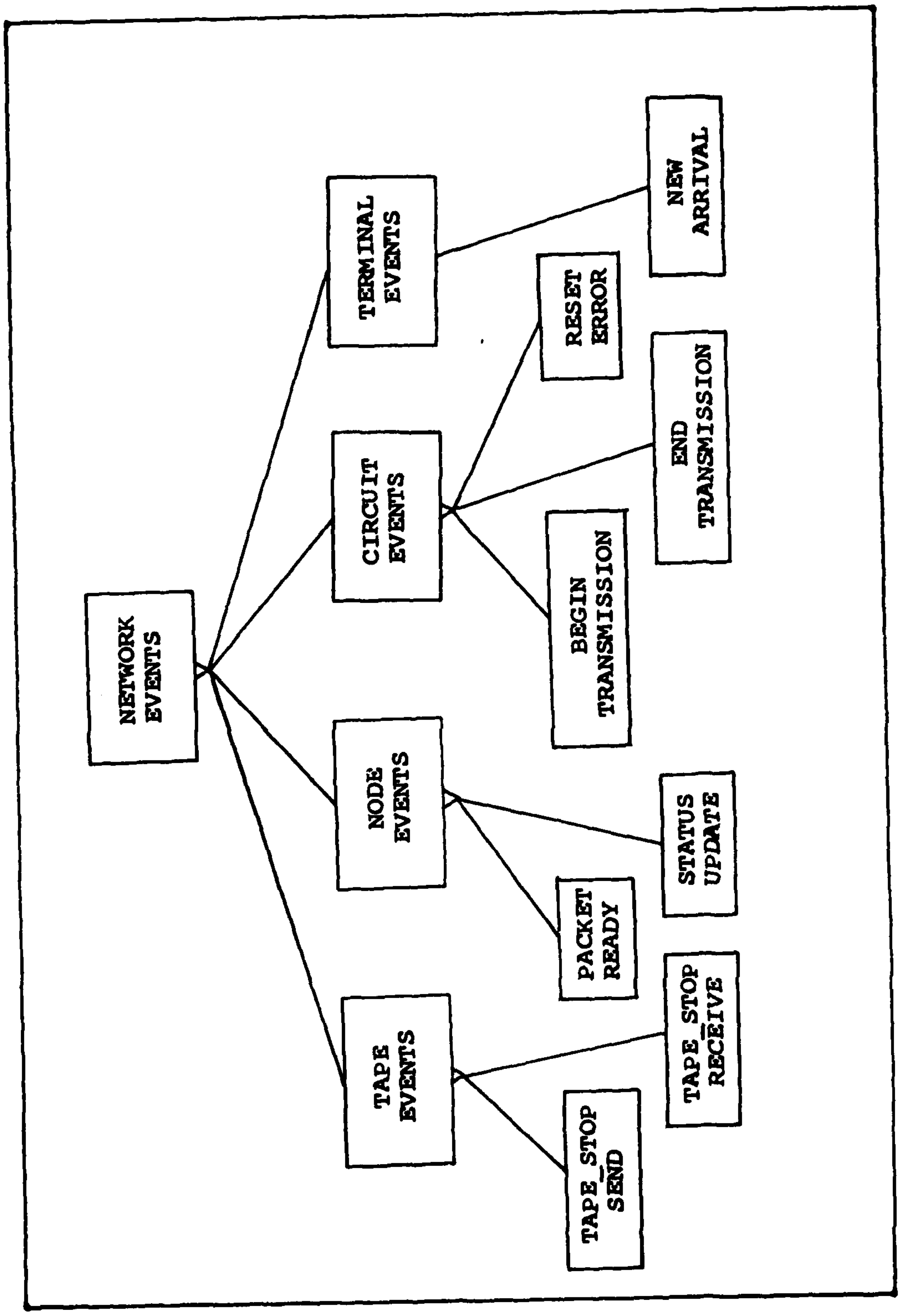

Figure 22. Network Events.

Tape Stop Send. A Tape_Stop_Send event occurs when a tape drive's 30 second Send_Control_Table timer runs out, indicating that it is time to retransmit the last two data blocks. A Tape_Stop_Send event should only occur if the network is heavily overloaded.

Tape Stop Receive. A Tape_Stop_Receive event occurs when a receiving node's two second Receive_Control_Table timer runs out, indicating that it is time to request a retransmission of the outstanding packets in the last data block.

Packet Ready. A special holding queue at each node is used to control the input of packets at a node. The Packet_Ready event signals that it is time to place a new data packet into the node's job queue.

Update Status. Update_Status indicates that it is time for a node to send status packets to its adjacent nodes.

Begin Transmission. Begin_Transmission signals that it is time to start a new transmission at the sending end of a circuit.

End Transmission. The End_Transmission event indicates that it is time to receive a transmission at the receiving end of a circuit. The begin and end transmission events could be combined into a single event if we were to assume that there is no delay between the end of one transmission and the start of the next. However, in this model we delay the start of the next packet transmission one millisecond to account for the time to link the next packet into the circuit.

Reset Error. A Reset_Error event occurs when a circuit's one second hold timer runs out, causing it to retransmit a burst of packets.

New Arrival. The New_Arrival event signals the arrival of a new terminal data packet at source node.

The events are processed using a set of data structures, which contain the necessary information and a set of procedures which specify the activities to be performed.

Data Structures

Each primary data structure is a composite of many individual variables and secondary data structures. The content of each structure is outlined below:

Nodes. A node is a record that contains:

1. The node's alphanumeric name.
2. The number of active ports.
3. A Reachable_Nodes table when the BDN routing algorithm is used or a minimum hops table and minimum plus one hops table when the minimum number of hops routing is used.
4. A job queue.
5. A packetizer.

The packetizer is a priority queue which is used to control the flow of packets into the node. All the packets originating at a node are placed in the queue. At discrete intervals, if space is available, the packets are moved one at a time to the job queue. A priority queue is an array of queues, one for each priority, in which each packet enters the rear of its appropriate queue and is removed from the front of the queue. Highest priority packets are always selected first. A job queue is a modified priority queue which is better suited for removing a packet from the middle of the list. In a normal queue each packet contains a single pointer to the next packet in the queue. In order to remove an item in the middle, the program must traverse the queue, starting at the front, until the packet just before the current packet is found. It then moves the predecessor packet's pointer from the current packet to the next packet in the list; thereby, deleting the current item from the queue. The speed with which this operation can be performed is dependent on the length of the queue. By adding an additional pointer to each packet in the queue, so that it has pointers to both its predecessor and successor, the requirement to traverse the list is eliminated and the operation can be performed in constant time.

<u>Circuits</u>. A circuit is a record that contains the circuit's:

1. Baud rate.

2. Destination node.

3. Return circuit's port number.

4. Circuit queue and a pointer to the packet being transmitted.

5. NEXTCSN and EXPCSN (used by the link protocols).

6. Status.

A circuit can be in one of four states. It can be idle, starting to transmit, ending a transmission or holding for an acknowledgement.

Tape Drives. A tape drive is a record that contains:

1. The tape's source and destination. The destination is represented by a set of locations to permit multi-site addressing.

2. A Send_Control_Table and a Receive_Control_Table used by the message protocols.

3. A delay table used to collect maximum, minimum and average packet and block delay times.

4. Two variables, MINPKTS and DELTAPKTS, used to generate random message lengths.

Message lengths are generated uniformly over the range MINPKTS to MINPKTS + DELTAPKTS, using the formula:

$$\text{Messagelength (in Packets)} = \text{MINPKTS} + \text{TRUNCATE}(\text{RANDOM} \times \text{DELTAPKTS}) \qquad (13)$$

where MINPKTS is the minimum block size, RANDOM is a random number between 0.0 and 1.0 and DELTAPKTS is equal to the maximum block size minus the minimum block size plus one.

The Send_Control_Table contains the block control number for the next block to be transmitted (NEXTBCN), the time when blocks NEXTBCN-2 and NEXTBCN-1 were put in the system, their unacknowledged destinations and the number of packets in each block. The Receive_Control_Table contains the expected block control number of the incoming block and a list of unacknowledged packets for blocks EXPBCN and EXPBCN + 1.

Terminals. A terminal is a record that contains:

1. The terminal's source and its destination.
2. An arrival rate, a minimum interarrival time and a maximum interarrival time used to generate Poisson arrivals using Equation 13.
3. The total number of packets delivered and the minimum, maximum and total delay time for these packets.

Packets. A variant record is used to represent the five packet types used in the model. All of the packets contain:

1. The packet's source, destination and priority.
2. Three pointers for linking the packet into the circuit and job queues.

3. An on-queue flag to indicate that the packet is linked to a circuit queue.

4. The port on which the packet arrived at the node. (A zero is used to indicate the packet originated at the current site.)

5. A CSN and an ACKCSN used by the link protocols.

6. A negative acknowledgement flag, if negative acknowledgements are used.

In addition to the above, each individual packet contains variant entries. Data packets contain a tape number, a Block_Control_Number, a Packet_Control_Number, an End_Of_Message flag used by the message protocols and the time the packet entered the system. Reply packets contain a tape number, a Block_Control_Number and a message set, which is used to acknowledge block receipts or request retransmissions. Terminal packets contain a tape number and an input time. Status and acknowledgement packets have no additional entries.

Event Time Tables. The event times for each device type are stored in a separate array. The Circuit_Time and Terminal_Time tables contain one entry for each device. The Node_Time table contains a Packet_Ready event time and a Status_Update event time for each node. The Tape_Time table is a square matrix indexed in both dimensions by a node number. The diagonal entries are the Tape_Stop_Send event times and the remaining entries are the Tape_Stop_Receive event times.

Procedures

The program structure is outlined in Figures 23a-c. The program consists of four basic processes which include: an initialization process, a Find_Next_Event/Process_Next_Event loop and a print results process. The loop terminates when the network clock reaches a predetermined stop time.

Initialization. In the initialization process, the user sets up the network by specifying:

1. The number of nodes, tapes and terminals.
2. The number of circuits attached to each node.
3. The baud rate, destination and return circuit's port number of each circuit.
4. The routing algorithm (BDN, MINHOP or MIN+1HOPS).
5. The destination and average number of packets of each tape.
6. The destination, mean interarrival rate and maximum and minimum allowable interarrival times for each terminal.
7. The transmission error rate.
8. The program stop time.

After this information has been read into the computer, the program calculates the routing selection vectors for each circuit. All three routing algorithms are implemented using the MIN+1HOP selection process, i.e., using the MIN+1HOP

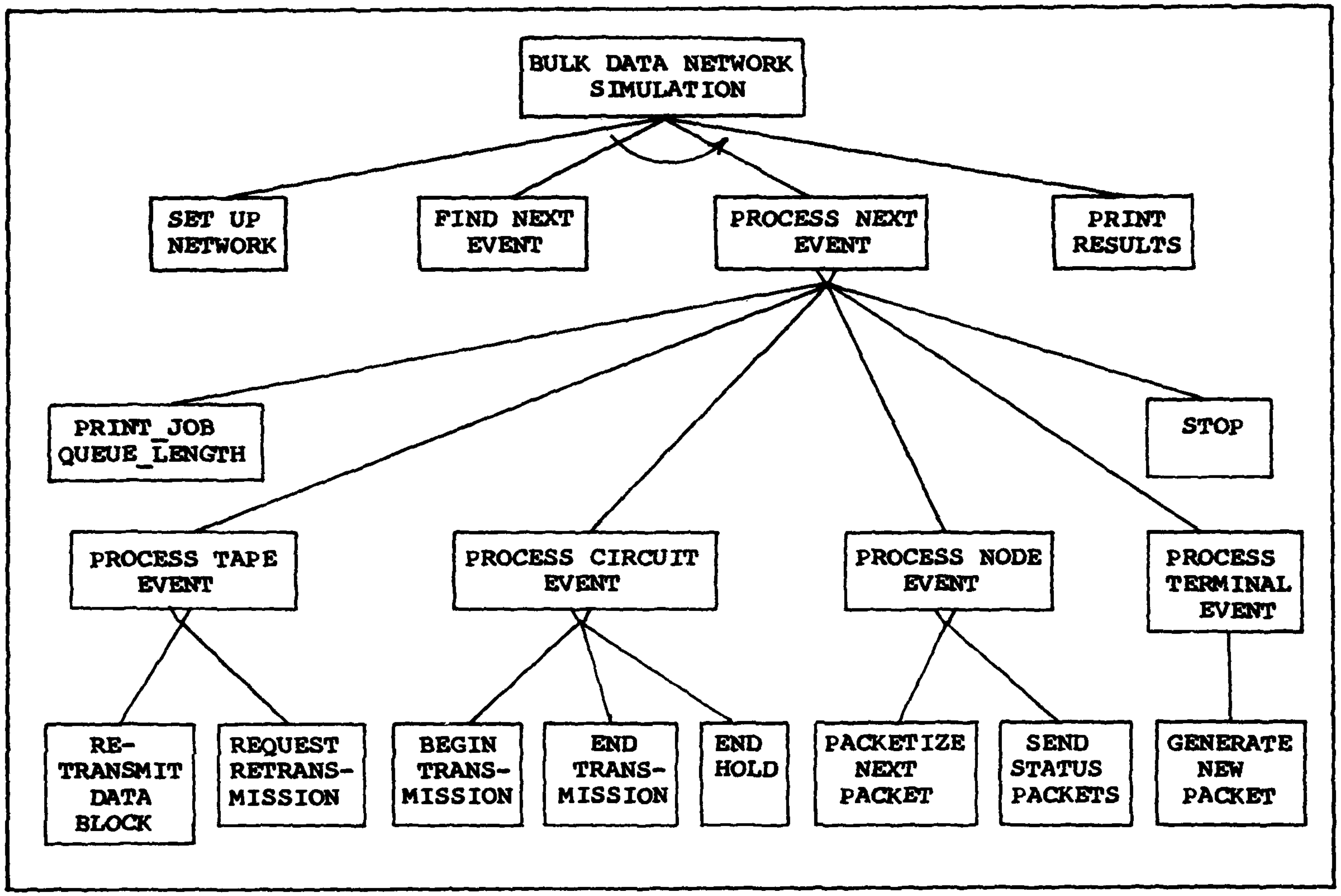

Figure 23a. Structure Chart.

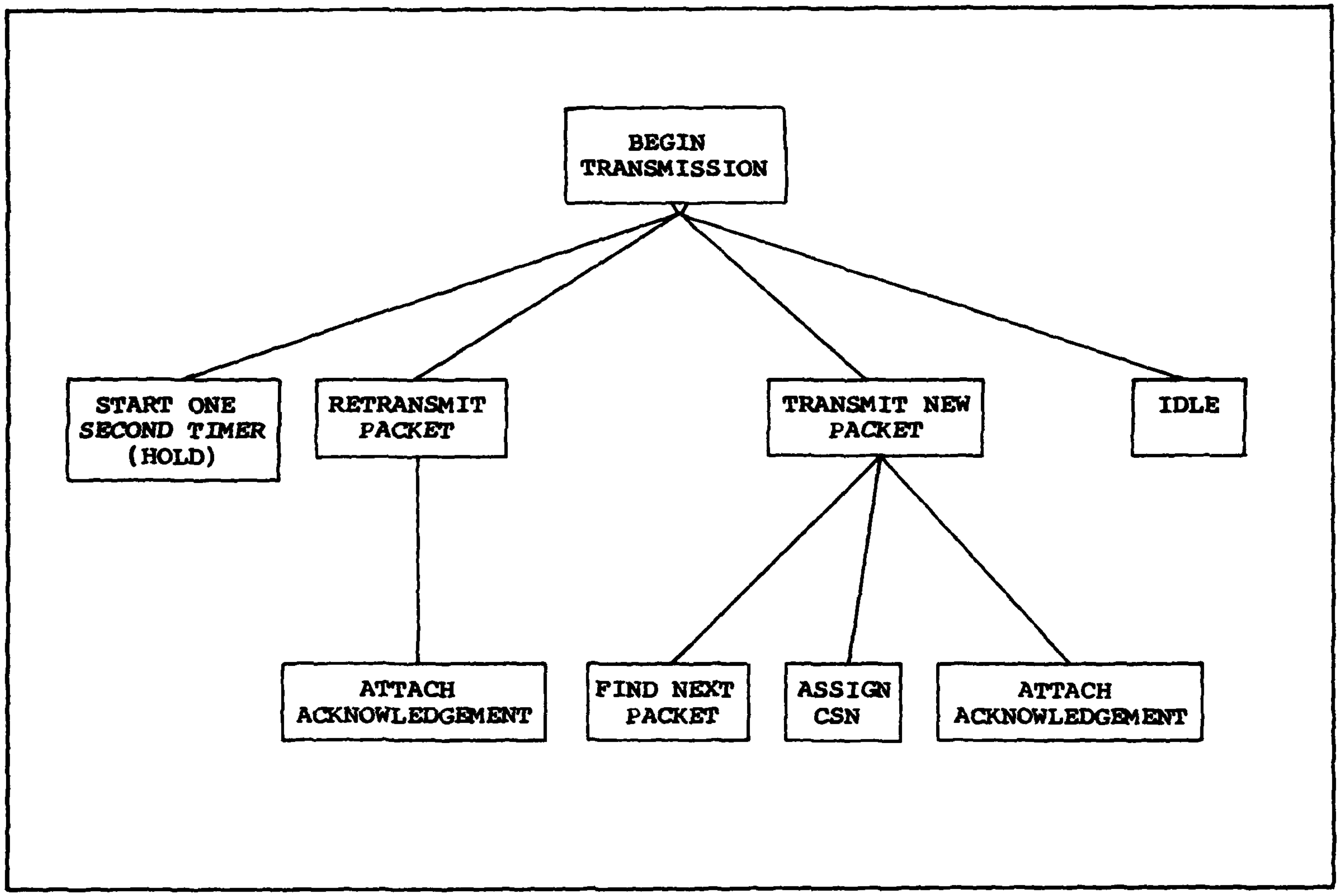

Figure 23b. Structure Chart (Continued).

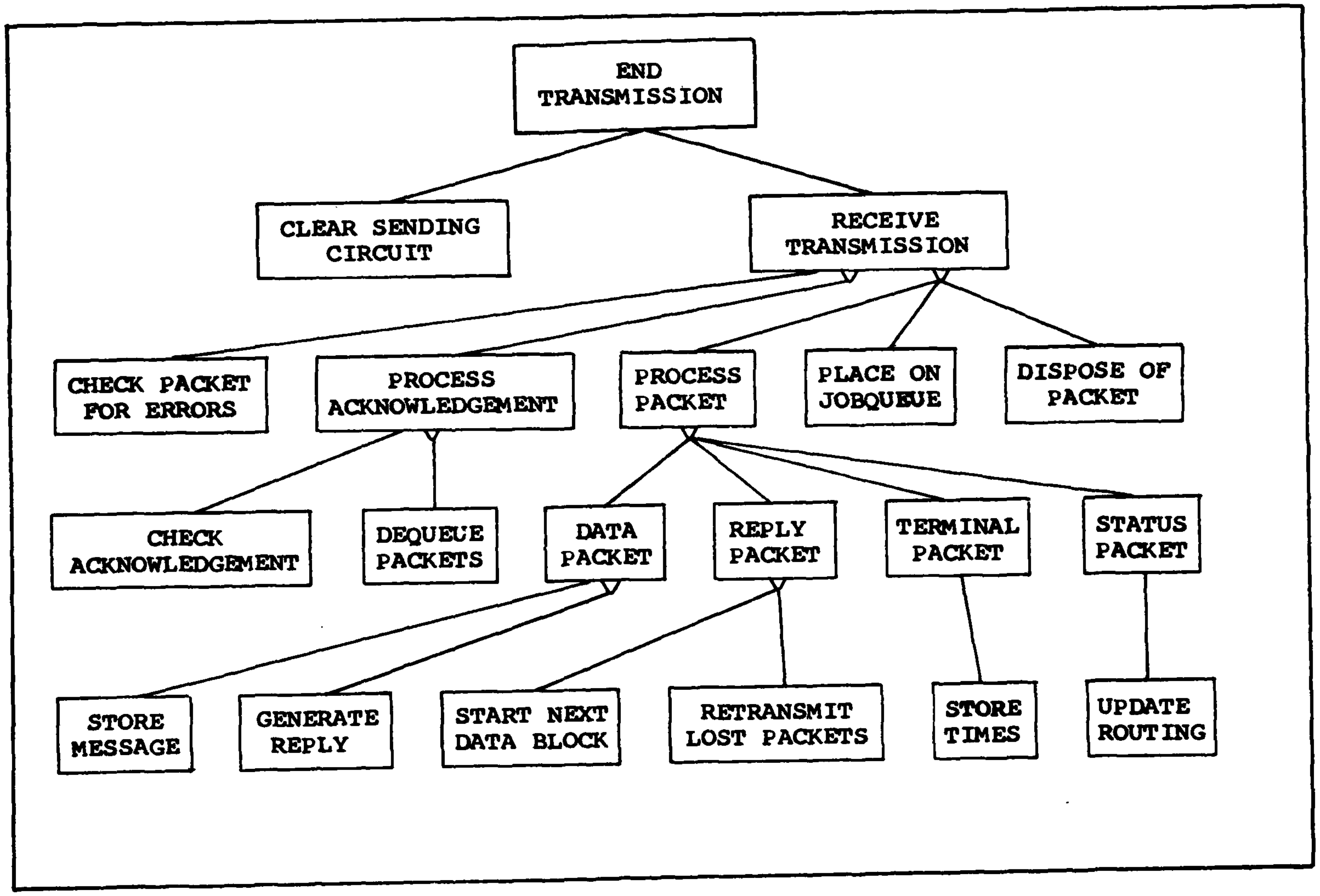

Figure 23c. Structure Chart (Continued).

selection vector, if the packet originated at the node and the MINHOP selection vector, if it did not. The different algorithms are implemented by calculating the appropriate values for each vector. If the current BDN method is used, both vectors will contain a list of the reachable nodes. If MINHOP is used, the MINHOP and MIN+1HOP selection vectors will contain the minimum hop selections.

Once the routing tables have been made, the program prints a table showing all the connectivity information. This is done so the user can check to see that the network has been specified correctly.

<u>Find Next Event</u>. The Find_Next_Event function is a search process which returns the next event to be processed along with the appropriate data structure indices needed to process the event. A site number is returned for a node event, a tape number for a tape event, a terminal number for the terminal event or a node number and a port number for a circuit event.

A linear search is used to find the next event. The time required to make the search is directly proportional to the total number of events (NE), which can be expressed by

$$NE = 2n + Pn + Tn^2 + In + 2 \qquad (14)$$

where

n is the number of nodes

P is the number of ports per node

T is the number of tapes per node

I is the number of interactive terminals per node

In Equation 14, 2n represents the total number of node events; Pn, the number of circuit events; Tn^2, the number of tape events; In, the number of interactive terminal events; and 2 is for the two overhead events, Stop and Print_Queue_Lengths. If we assume that P, T and I are limited to a maximum value for each node, then the search time can be expressed as a function of n. As n increases, the Tn^2term will dominate the others and, therefore, the search is order n^2.

The search can be improved by noting that if the network is not overloaded, the terminal events will hardly ever occur and that changes to their event times occur only at the source and destination nodes. Therefore, if we assume that the path length from the source to the destination is order n, then the tape event times will be changed only once in every n events. By searching the tape event table only when a change is made and storing the minimum event information in a separate location, the search process can be improved to O(n).

Process Time Out Send. The Time_Out_Send event is processed by the Retransmit_Data_Block function using the tape drive's Send_Control_Table. Block numbers, NEXTBCN-2 and NEXTBCN-1, are repacketized and the packets are addressed to all unacknowledged destinations. The new packets are placed in the source node's packetizer. The Retransmit_Data_Block function is used to start the first two blocks in a tape. After the event is processed, the event time is set to maximum, and a new minimum tape event time is found.

Process Time Out Receive. The Time_Out_Receive event is processed by the Process_Receive_Time_Out function using the tape's Receive_Control_Table. A reply packet, which contains a list of the outstanding packets in block number EXPBCN, is generated and placed in the receiving node's packetizer. Finally, the event time is set to maximum and a new minimum tape event time is found.

Process Packet Ready. The Packet_Ready event is processed by the Process_Node_Stop function which moves the highest priority packet in the node's packetizer to the node's job queue. If there is an idle outgoing circuit, which can accept the new packet, the circuit's status is changed to Begin_Transmission and its event time is set to the current time. If the packetizer contains other packets, its event time is increased by the time required to make a new packet, or, if this packet was the last in a block, by the time required to read a new block; otherwise, it is set to maximum. When the last packet in a block is placed in the job queue, its 30 second Send_Control_Table timer is started.

Process Status Update. The Status_Update event is also processed by the Process_Node_Stop function, which generates a short status packet for output on each circuit. Since we do not allow the connectivity to change during a simulation run, there is no need to transfer connectivity information. However, the packets are still transmitted because they represent an overhead transmission time which must be accounted for. Like other packets, the status packets are placed into the node's packetizer. The status event time is set to the current time plus 12 seconds.

Process Begin Transmission. The Begin_Transmission event is processed by the Process_Start_Transmission function (Figure 23b), which causes one of four activities to begin. If the circuit queue is full (contains four packets), an acknowledgement packet is generated at the receiving node, the circuit is placed in a hold status and a one second timer is started. If the current packet pointer is pointing to a packet which indicates that the packet must be retransmitted, an acknowledgement is embedded in the packet and the circuit's status is changed to End_ Transmission. The circuit's next event time is set to the current time plus the packet's transmission time. If the current packet pointer is pointing to nil, the circuit requests another packet from the job queue using the Find_New_Packet function. The new packet is assigned the NEXTCSN, and ACKCSN is attached and the packet is transmitted. If a new packet cannot be found and the circuit queue is empty, the circuit status is changed to idle and event time set to maximum. If the circuit queue is not empty, the circuit is placed in a hold status and the event time set ahead one second.

Process End Transmission. The End_Transmission event is processed by the Process_End_Transmission function (Figure 23), which clears the sending end of the circuit and processes the incoming packet at the receiving end. The sending circuit is cleared by moving the current packet pointer to the next packet in the circuit queue. If the current packet is being transmitted for the first time, it will be the last item in the circuit queue. As a result of this operation, the current packet pointer will point to nil. Next, the circuit's status is changed to Begin_Transmission and the event time is set ahead one millisecond to account for the node's processing time. A random error generator is used to check the incoming packet for errors. If an error is found, the packet is discarded and a NAK, if they are used, is generated on the return circuit. If the packet was correct, the packet CSN is checked against the EXPCSN; if they match, the packet is received and acknowledged. Next, the ACKCSN is stripped off the packet and checked. If it is valid, the designated packets are removed from the return circuit's circuit queue. If this circuit was in a hold status, the status is changed to Begin_Transmission and the circuit restarted. If the incoming packet was destined for this node, the message is processed and the packet is either routed on to other destinations or discarded.

The type of message processing performed depends on the packet's type. If the packet is the first data packet in a block, a new RCT is set up and the packet's control number is checked off. As subsequent packets arrive, their PCN's are deleted from the list and their delay times are recorded in the delay table. When the last packet in the block arrives, the RCT is checked to see if all the packets are in; if so, a reply packet with an empty message set is generated to acknowledge receipt of the block. If some packets are still outstanding, the tape's two second receive control timer for this node is started. Reply packets are processed by transmitting the next block or retransmitting the packets requested in the message set. A random message length function is used to determine the number of packets in the new block. If the packet is a terminal data packets, the delay time is calculated and recorded. In the real network, status packets could dictate changes to the routing tables; but in this model, where the connectivity is fixed, they cannot produce changes and therefore, we simply discard them.

Process Retransmit Error. The Retransmit_Error event is processed by setting the circuit's packet pointer to the first packet in the circuit queue and starting a new transmission.

Process New Arrival. The New_Arrival event is processed by generating a new terminal data packet and placing it in the source node's packetizer. The packet's arrival time is recorded in the packet. Finally, the time for the next arrival is calculated using Equation 12 presented in Chapter IV.

Process Print Results. When the simulation terminates, the following information is printed:

1. Job queue length for each node.

2. The total number of packets and blocks transmitted by each tape. The maximum, minimum and average delay time for each and the transmission rate in packets/hour and block/hour for each tape.

3. The total number of terminal data packets for each terminal along with the maximum, minimum and average delay times for those packets.

Summary

The program described in this chapter provides a set of four basic network building blocks which can be linked together to model the Bulk Data Network. The model can be used to study the performance of the network on both batch and interactive traffic for various error rates. The user can specify either the current routing algorithm, the MINHOP algorithm or the MIN+1HOP algorithm. He also specifies the network connectivity, number of tapes, number of terminals and whether NAK's are to be used.

Once the design is complete, the next step in the simulation process is to implement and test the program.

VI. Implementation and Testing

The program was coded in PASCAL and run on the AFIT CDC CYBER computer. It uses only 45K words of memory and can either be batch processed or run interactively using INTERCOM. A typical simulation with ten input tapes takes 60 seconds of CPU time to produce 240 seconds of simulated time. The program listing and a user's manual are given in Appendices B and C.

The completed program was tested in stages, beginning with a simple model and then adding features until the entire program had been examined. The objectives of this effort were to verify that the program correctly implements the network activities identified in the function analysis and that the model's results are compatible with those observed in the actual network.

Verification

Program testing is basically concerned with two concepts; data values and decision paths, which map directly into the concepts of data structures and procedures discussed in the last chapter. For the program to work properly, the individual variables in each data structure must be limited to a meaningful range of values. (e.g., minus one would not be a meaningful value for the number of nodes.) The program should return valid results for all values within the specified range, and notify the user when a variable is out of range. Since most errors occur for values at the ends of the range, a testing technique called boundary analysis was used. This technique calls for running several sample values, which include the boundary values for each variable, a random value inside the range and a random value outside the range. The concept employed is that if the program works at the boundaries and it works for a random value inside the range, then it should work for all values inside the range. The same is true for detecting values outside the valid range. PASCAL allows the programmer to specify the range for each variable. This process is referred to as strong typing. The use of strong data typing greatly simplifies the testing process. An attempt to assign a value outside the valid range will cause the computer to abort the run and identify the error. In addition, by specifying the valid range, the programmer identifies the boundary values that need to be tested.

Strong data typing will not catch all errors because there are situations in which the combination of valid values produces an invalid value. An example of this is the circuit connectivity. Each circuit's return circuit must be correctly identified for the link protocols to work. It is possible for the program user to specify the wrong return circuit, while giving a legitimate circuit number. The program is not designed to catch this type of error, therefore, the input data is processed and the connectivity information is displayed on the output before the simulation begins. It is the user's responsibility to ensure that the configuration is correct.

The program functions can be viewed as a series of roads with a number of intersections. The program moves forward until an intersection is reached; it must then choose one road or the other and then proceed to the next intersection. Each unique path from the start of the journey to the end is a decision path. Decision path testing is a process that involves running a set of test values that will cause each decision path to be taken at least once. The decision paths are identified from the activity and control specifications in the functional description.

There are basically three types of decisions; those associated with a source node, those associated with an interim node and those associated with a terminal node. Every node will make each type of decision at one time or another. Since the program was written in modules, where each node is modeled by the same set of procedures, we only have to test one node to be certain that the process works for all nodes.

The testing process was begun with a simple two node network with one input tape (Figure 24). The message length was held constant and no transmission errors were allowed. The program was augmented by adding a number of print routines

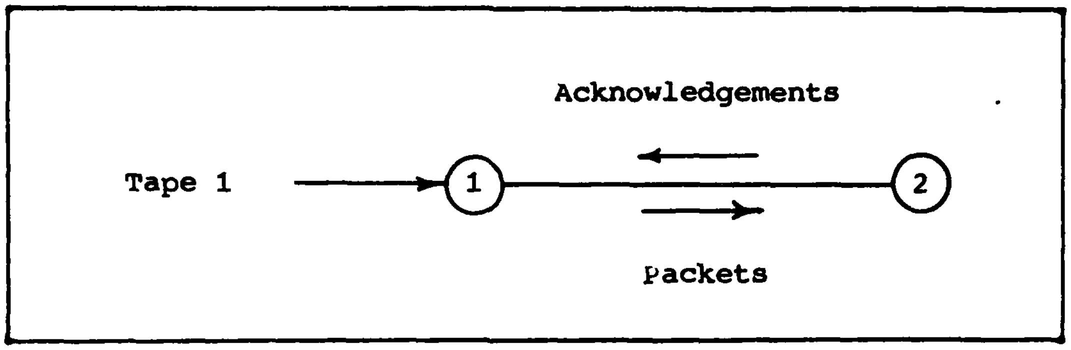

Figure 24. Test Network 1.

that printed the current value of key program variables as the program stepped through the simulation. This simple network can be modeled deterministically. Each packet will take 292 milliseconds to transmit. After every fourth packet is transmitted, the transmission is stopped until an acknowledgement is received from the destination. The acknowledgement takes 27 milliseconds.

This simple model allowed us to observe:

1. The generation of packets by the tape drive.
2. The placement of packets into the packetizer and then into the job queue.
3. The selection of packets from the job queue (routing decision).
4. The transmission process.
5. Valid packet acknowledgement.
6. Complete block acknowledgement.

Next, a transmission error was induced in the transmission process. The network responded correctly by retransmitting the bad packet after a one minute hold. Then, a packet's sequence number was deliberately altered to test the message protocols. The program correctly identified the packet as being a duplicate and discarded it. After the last packet in the block arrived, the two second receive control timer was started, and when the missing packet did not arrive in two seconds, a reply packet was generated. Upon receipt of the reply, the packet was retransmitted, this time with the correct PCN. When the packet arrived, the block was successfully acknowledged. The same process was used to test the send time-out function. The last packet in the first block was misidentified and therefore, not acknowledged by the receiving node. After 30 seconds, the first two blocks were retransmitted.

Next, a second tape drive was added to the network at Node 2.

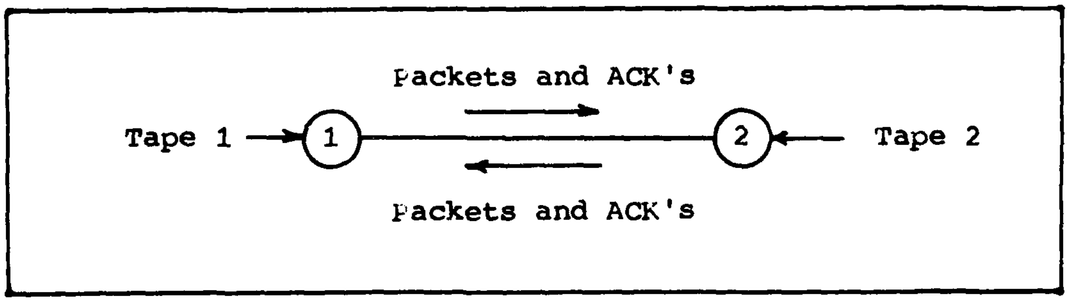

Figure 25. Test Network 2.

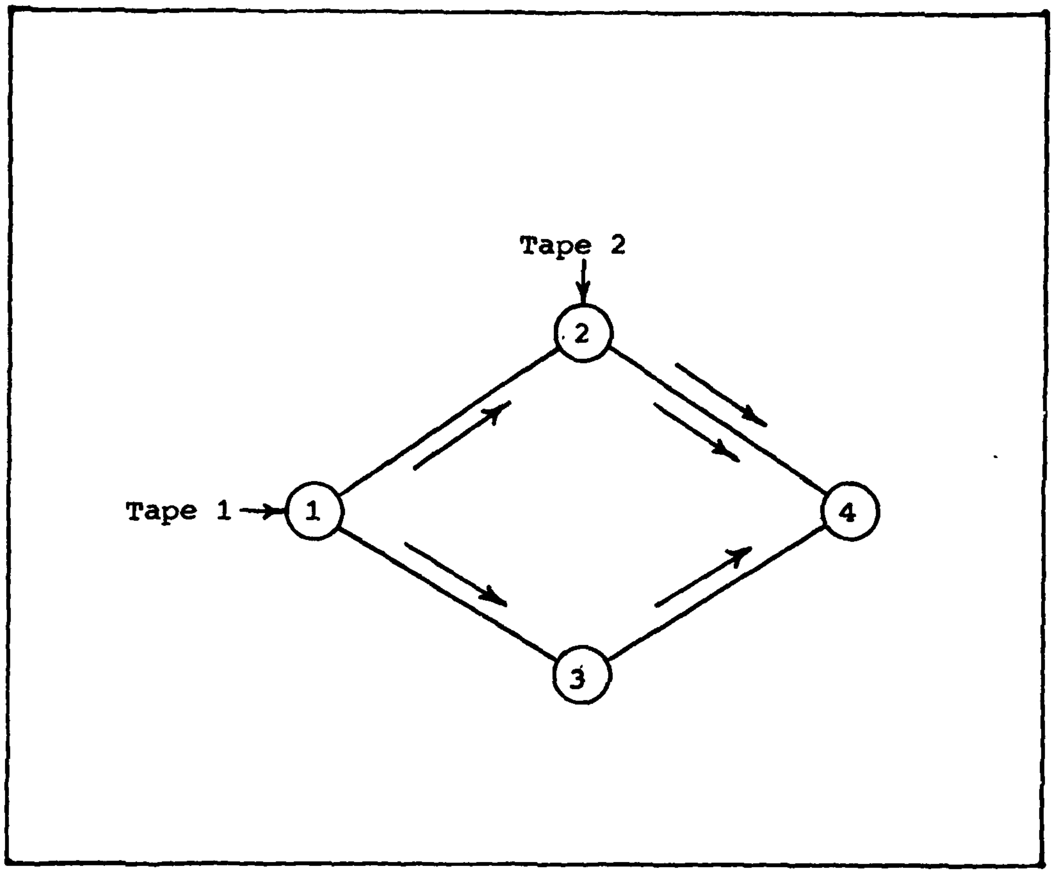

Figure 26. Test Network 3.

This configuration was used to observe the embedding of acknowledgements in packets going the other way.

The network was then enhanced by adding two more nodes to produce a diamond structure (Figure 26). This configuration was used to test the transmission of a packet to multiple sites (Node 1 to Nodes 2 and 3, and Node 1 to Nodes 2, 3, and 4), and the interim node decision processes. The addition of a second tape drive at Node 2 identified an important network effect. If both Node 1 and Node 2 are attempting to transmit a tape to Node 4 at the same time, Tape 2 will block packets from Tape 1. This is caused by Tape Drive 2 placing packets into the job queue much faster than Tape 1 can transmit them from Node 1. Tape 2 dumps an entire block of packets in the queue at one time. It then starts processing the next block. While it is processing the block, packets are arriving from Node 1 and are being placed in the job queue. When Tape 2 has finished processing the block, it dumps another batch of packets in the queue. The degree of interleafing between the packets is dependent on the time for Node 2 to generate a block of packets, the number of packets in the block, and the rate of packet arrivals from Node 1. It generally takes about a half second to process the block and a quarter second to receive a transmission. Therefore, two incoming packets will be sandwiched between each block generated at Node 2. If the number of packets in the block is small, only a small delay occurs; as the block size increases, so does the delay for Tape 1 packets.

The terminal packet operation was tested separately from the tape activity. The simple two node network shown in Figure 27 was used.

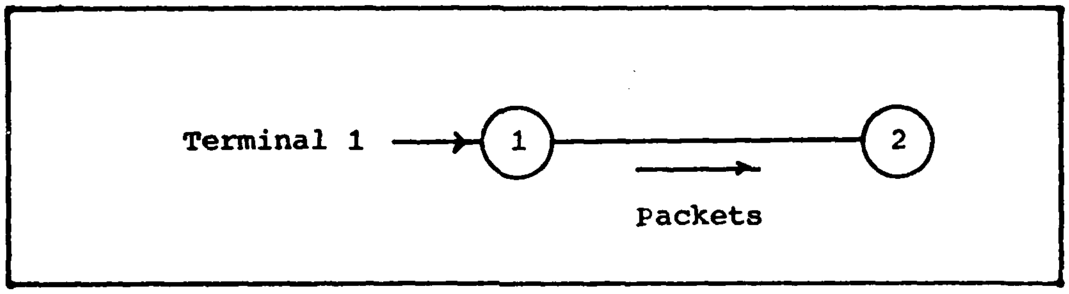

Figure 27. Test Network 4.

This network can be modeled using a very simple single server queueing model. According to Schwartz (13) the average packet delay is given by

$$E(T) = \frac{1}{\mu - \lambda} \tag{15}$$

where λ is the average arrival rate and μ is the service rate, which is the reciprocal of the average packet transmission time. The model was run for ten values of λ .

The model results are displaced along with the theoretical results in Figure 28. The observed packet delay curve has the characteristic exponential shape. The variance between the two curves is due to the fact that the queueing model shows an average result based on all possible input values and exponential service times, while the actual curve is based on a small random sample with two service times based on 16 character and 174 character

AD-A115 552 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
SIMULATION AND ANALYSIS OF THE AFLC BULK DATA NETWORK USING ABS--ETC(U)
DEC 81 S D STEWART
UNCLASSIFIED AFIT/GCS/EE/81D-15 NL

2 OF 2
AD
A115552

END
DATE
FILMED
7-82
DTIC

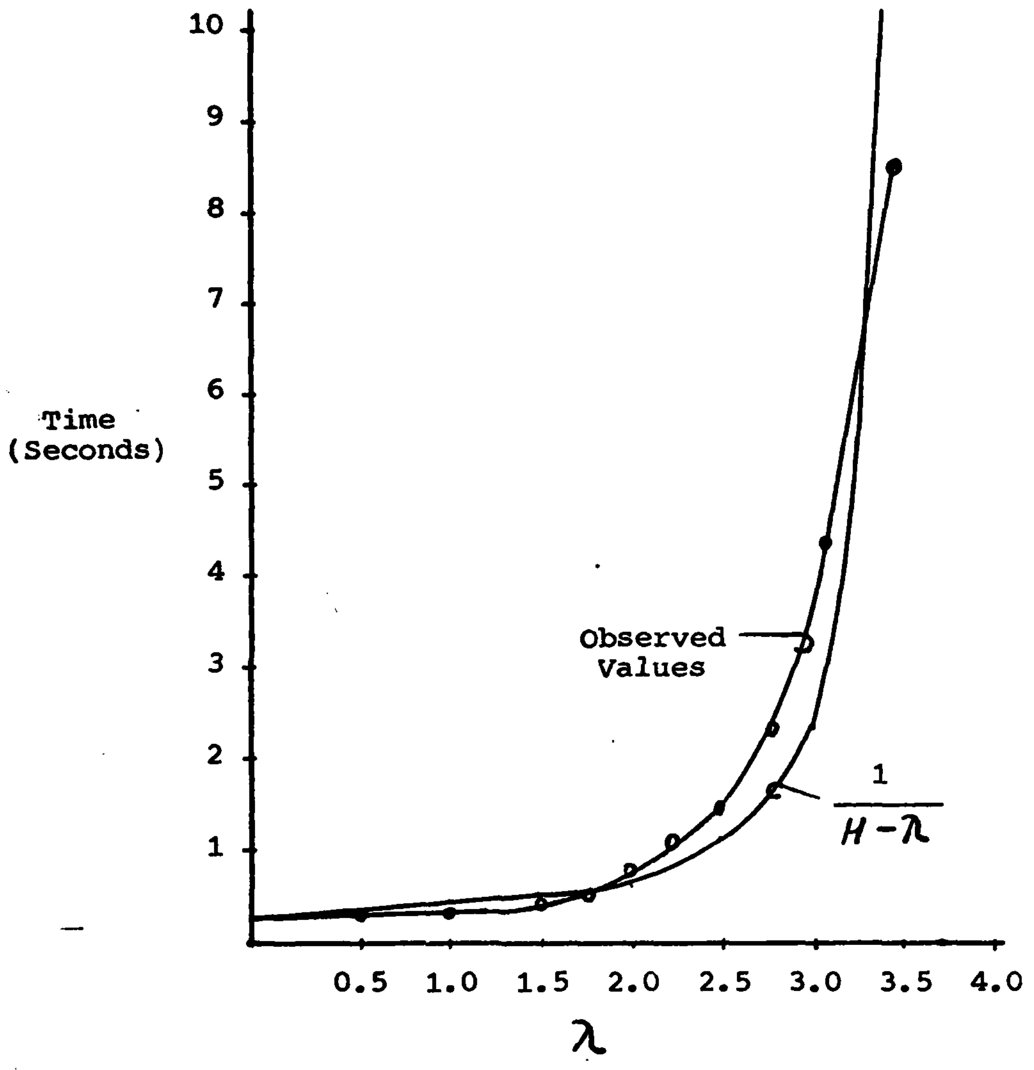

Figure 28. Average Packet Delay.

packets. The queue length at Node 1 was growing at the end of the last simulation which indicates the network is overloaded. The theoretical curve goes to infinity at this point.

Validation

The model was validated by running a sample set of test data on both the network and the simulation and comparing the results. Two different tapes were used for the test. The first was a test tape that contained 181,240 data characters in 503 blocks and the second was an actual working tape consisting of 306,150 characters in 106 blocks. The system compressed the first tape into 1,203 packets, (an 8% reduction) and the other tape into 996 packets (a 53.2% reduction). A detailed breakout of each tape is given in Table I.

Thirteen separate test runs were made. The first six runs involved the transmission of a single tape from one node to an adjacent node. The next four runs involved the transmission of a single tape from one node to a nonadjacent node. The third set of two runs, consisted of the the transmission of one tape to both an adjacent and a nonadjacent node at the same time. The last run was the transmission of two tapes simultaneously from Node 3 to all other sites.

During each test run, SKDAR recorded the total workload on each circuit and the transmission error rate. The transmission time for each run is shown in Table II. The

TABLE I

Test Tapes

Tape 1

50	blocks	x	40	characters	=	2,000
53	blocks	x	80	characters	=	4,240
100	blocks	x	150	characters	=	15,000
200	blocks	x	300	characters	=	60,000
100	blocks	x	1,000	characters	=	100,000
503	blocks					181,240 characters

Tape 2

3	blocks	x	80	characters	=	240
103	blocks	x	2,970	characters	=	305,910
106	blocks					306,150 characters

TABLE II

Validation Test Results

Test Number	Tape	Source	Destination	Blocks	Packets	Transmission Time (min:sec) Network	Transmission Time (min:sec) Simulation
I-1	1	1	2	503	1203	5:19	5:26 *
2	1	1	2	106	996	4:38	4:42
3	1	4	5	503	1203	5:28	5:26 *
4	1	4	5	106	996	4:36	4:42
5	1	2	4	503	1203	5:00	5:00
6	1	2	4	106	996	4:36	4:42
II-1	1	5	6	503	1203	5:46	5:59
2	1	5	6	106	996	2:45	2:28
3	1	2	6	503	1203	6:03	5:59
4	1	2	6	106	996	2:48	2:28
III-1	1	1	3,5	503	1203	6:56	7:06
2	1	1	3,5	106	996	4:48	4:44
IV-1	1	3	1,2,4,5,6	503	1203	12:30	-
	2	3	1,2,4,5,6	106	996	5:55	5:58

* With 1% Error Rate

additional traffic on the SM-ALC to OO-ALC circuit forced most of the test packets to take the Site 1-3-5 path. A transmission error rate of 1% was recorded on two of the runs.

The thirteen test runs were simulated on the model network. The program was modified to stop when a specified number of blocks on a designated tape had been transmitted and acknowledged. Since the last packet in a data block does not have to be a full packet (144 data characters), an estimate of the last packet's size had to be made. The theoretical single hop transmission time for 1,203 packets, assuming no wasted time, is 357 seconds. The best observed time was 300 seconds, therefore, the first test tape had to contain at least 19% fewer characters. The variance for the second tape was only 3-4%.

In the model, a step function was used to generate the message lengths for each block in Tape 1. The first 203 blocks contained one packet each, the next 200 blocks contained two packets and the last 100 blocks contained six packets each. The packet size was reduced to 140 characters to take into account the short packet variance. The second tape was modeled by a tape with nine packets per block. The packet size was not changed since the modeled tape contains 4% fewer packets.

The last simulation run was stopped when the second tape was completed; therefore, no time was obtained for the first tape. The simulated and actual runs compared reasonably well. All but two were within 2% and the others were within 10%. The existence of background traffic can account for the variation.

Summary

The program was tested in a top-down fashion beginning with the basic network activities and then, repeatedly adding features until the entire program had been tested. Next, the program was used to model the existing network and a series of 13 test runs were made using two significantly different test tapes. The same runs had been previously run on the actual network and therefore, a comparison of the results could be made. Based on the closeness of the results, we can assert that the program is both correct and valid, and therefore, can be used to simulate alternative network configurations. The results of several such simulations are discussed in the next chapter.

VII. Results of the Simulation

The validated model was used to examine the connectivity, routing, interactive terminal and negative acknowledgement proposals presented in Chapter IV. The three routing algorithms were evaluated on both the current network and the upgraded network. The interactive terminal additions were evaluated on the upgraded network and the negative acknowledgement proposal was evaluated using both a simple two node configuration and the upgraded network. This chapter presents the findings of each study.

Additional Circuits and Routing Algorithms

The network was upgraded by adding the OO-ALC to HQ AFLC and new SM-ALC to OO-ALC circuits to the existing network. Six sample workloads were run on the current network and then the upgraded network to determine the impact of the increased capacity on block throughput. Each test was repeated three times for a simulated 240 seconds, using a different routing algorithm on each run.

The number of input tapes in a run was varied from three to twelve in order to simulate light, medium and heavy loading. The source and destination(s) of each tape is shown in Table III. During Run 6, each input tape had an average of ten packets per block. The average block size was allowed to vary from tape to tape in the other

TABLE III

Test Data

Test Case 1

Number	From	To	Block Size
1	1	2	5.0
2	2	3	4.5
3	5	6	8.0

Test Case 2

Number	From	To	Block Size
1	1	4	8.0
2	1	3	4.0
3	3	1	5.0
4	6	5	5.0
5	2	1	6.0

Test Case 3

Number	From	To	Block Size
1	1	3,5	5.0
2	2	3	5.0
3	3	1	6.5
4	4	6	8.0
5	5	3	5.5
6	6	5	7.0

Test Case 4

Number	From	To	Block Size
1	2	1,3	3.5
2	4	5	5.0
3	1	2	4.5
4	1	3	8.0
5	6	2	5.5
6	6	4	2.5
7	3	5	8.5
8	5	4	6.0
9	5	6	3.5
10	5	3	4.5

Test Case 5

Number	From	To	Block Size
1	2	1,3	4.0
2	4	5	7.5
3	1	2	4.5
4	1	3	8.0
5	6	2	6.0
6	6	4	2.5
7	3	5	8.5
8	5	4	8.0
9	5	6	3.5
10	5	3	4.0
11	4	5	6.5
12	3	2	5.0

Test Case 6

Number	From	To	Block Size
1	1	2	10.0
2	1	5	10.0
3	4	2	10.0
4	5	4	10.0
5	5	6	10.0
6	6	1,4	10.0
7	2	1	10.0
8	2	6	10.0
9	2	5	10.0

runs to provide a more realistic mix of jobs. The results of each of these runs were normalized to ten packets per block by dividing the tapes' average block size by ten and multiplying the result by the number of tapes transmitted. Block lengths were allowed to vary within one or two packets of the mean.

The results of the simulations are summarized in Table IV. The length of the job queues did not increase toward the end of each run; this indicates that the network was stable. The new circuits increased the total block throughput by 30.5% when the BDN routing algorithm was used, by 33.5% when the MIN+1HOP routing algorithm was used, and by 34.7% when the MINHOP algorithm was used. In the present seven circuit configuration, the MINHOP algorithm outperformed the BDN method by 5.8% and the MIN+1HOP method by 8%. When the network was upgraded, the MINHOP algorithm outperformed the others by 8.8 and 8.9% respectively.

In the current configuration, the BDN and MIN+1HOP circuit selection vectors are nearly the same, and as a result, many of the runs produced identical results for the two methods. However, for other loadings, the differences between the selection vectors is significant. This was demonstrated in Run 6.

When the OO-ALC to HQ AFLC circuit is added, more nodes are adjacent to one another and the BDN selections become more restrictive than the MIN+1HOP selections. In heavier

TABLE IV

Normalized Test Results

(Total Blocks Transmitted)

TEST RUN	CURRENT NETWORK			UPDATED NETWORK		
	BDN	MINHOP	MIN+1HOP	BDN	MINHOP	MIN+1HOP
1	270	312	270	388	419	392
2	431	437	431	423	563	436
3	380	431	372	387	468	433
4	543	537	543	784	784	767
5	595	594	595	851	871	844
6	604	677	554	866	920	821
TOTALS:	2,823	2,988	2,765	3,699	4,025	3,693

loadings, the BDN method generally outperformed the MIN+1HOP method, parallelling the performance of the MINHOP method.

There are two opposing factors which account for the variance between the routing methods. First, the less restrictive algorithms, BDN and MIN+1HOP, provide more unique paths for a packet to follow in reaching its destination and thus increase the effective circuit capacity between the two nodes. If there are two paths between a pair of nodes, which do not share a common circuit, then data can move twice as fast as it could if there was only one path between them. However, if the transmission of packets on either path is blocked at an interim node, the overall block throughput can be hurt. If one packet in a block is held up, the entire block is held up until the delayed packet is either released by the interim node or a repeat packet is received on another path. The MINHOP algorithm minimizes the probability of a packet being blocked by providing fewer path choices. In this study, blocking was found to be the most significant factor.

The impact of packet looping on network throughput was examined using the upgraded network with the SA-ALC to WR-ALC circuit deleted. As stated previously, the BDN routing method will not prevent packets sent from SM-ALC to WR-ALC from looping, given this connectivity. A tape, with five packets per block, was transmitted from Node 1 to Node 5 and an identical tape was transmitted in the

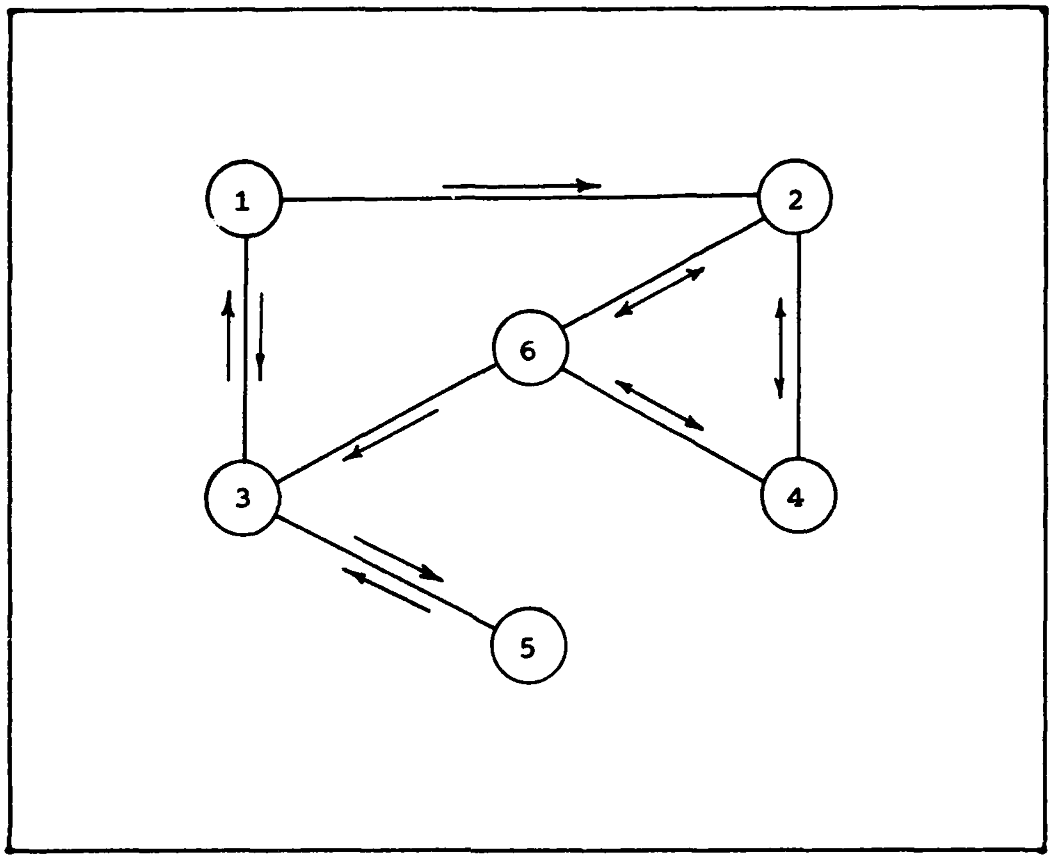

Figure 29. Paths from Node 1 to Node 5.

reverse direction. When the MINHOP or MIN+1HOP routing algorithms were used, 160 blocks were transmitted both ways in four minutes. When BDN routing was used, only 99 blocks of the SM-ALC to WR-ALC tape and 132 blocks of the WR-ALC to SM-ALC tape were transmitted. Even though the WR-ALC to SM-ALC path does not contain a loop, the tape was delayed because the replies were being lost in the loop on the return trip (Figure 29).

Negative Acknowledgements

The use of negative acknowledgements to request the retransmission of bad packets was examined first with a simple two node network, and then with the upgraded BDN. The two node model contained a single input tape at Node 1 that was transmitted to Node 2. The simulation was run for 240 seconds with error rates of 0, 1, 3, 5, 7.5 and 10%. The total block throughput is shown in Figure 30. The use of negative acknowledgements substantially improved the total throughput.

Next, Test Case 6, with nine input tapes, was rerun on the upgraded BDN model using MINHOP routing with error rates of 1% and 3% (Table V). When NAK's were used, throughput fell by 3% at the 1% error level and by 9.7% at the 3% error level. Without NAK's, the throughput fell by 13% at the 1% level and by 28.7% at the 3% level.

Interactive Terminals

The last step in the analysis of the network was to evaluate the interactive terminal proposal. The objective of the effort was to determine if the network could provide a maximum response time of less than two seconds for a round trip transmission. The proposal was evaluated on the upgraded network using both the BDN and MINHOP routing algorithms. The upgraded network was used because the new circuits will be installed before the terminals would be added.

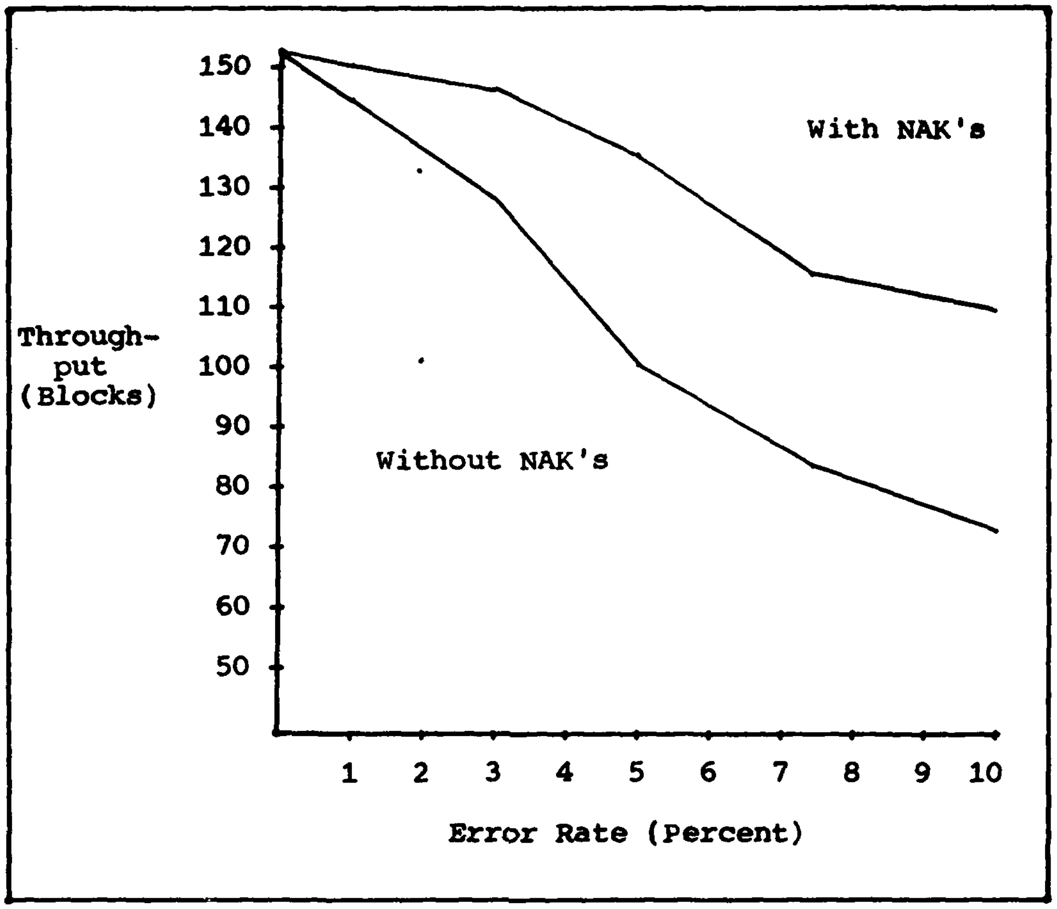

Figure 30. Impact of NAK's on Throughput.

TABLE V

Impact of NAK's on Test Case 6

Error Rate (%)	Throughput (Blocks)	
	With NAK's	Without NAK's
0	920	920
1	888	799
3	831	656

Since the Poisson distribution is characterized only by its mean, the five terminals at WR-ALC and SM-ALC can be modeled using a single terminal with a mean arrival rate of 5 λ , where λ is the average arrival rate for each terminal. Assuming 1,000 packets per day, per terminal, and an eight hour work day, the composite arrival rate is 0.17 packets per second.

The network was modeled by placing one composite terminal at WR-ALC and another at SM-ALC to provide the originating traffic, and two terminals at HQ AFLC to provide the return traffic. The upper and lower bounds for the packet interarrival times were 25 to 0.01 seconds. Background tape traffic was provided by four tape drives placed in the same locations as the terminals. This was done to produce the worst case test conditions (i.e., busy circuits). The results for the first two runs using the current routing algorithm and the MINHOP algorithm are shown in Table VI. Neither run provided a maximum roundtrip time of less than two seconds, even though both methods had an average time well below this figure. Even when the tape traffic was removed, the slowest roundtrip time for WR-ALC to HQ AFLC was still over 2.5 seconds (Table VI). The minimum transmission time between WR-ALC and HQ AFLC is 0.582 seconds. With a return trip, the minimum time is 1.164 seconds; therefore, is two or three packets arrive close together, the queueing delay causes the maximum delay to go over two

TABLE VI

Terminal Packet Delay Times (Milliseconds)

	SM-ALC to HQ AFLC		HQ AFLC to SM-ALC		WR-ALC to HQ AFLC		HQ AFLC to WR-ALC	
	Average	Maximum	Average	Maximum	Average	Maximum	Average	Maximum
Upgraded Network (With Background Traffic)								
BDN	0.478	0.957	0.596	1.549	0.733	1.129	1.015	2.212
MINHOP	0.539	1.291	0.704	1.880	0.815	1.505	0.798	1.062
Upgraded Network (Without Background Traffic)								
MINHOP	0.307	0.582	0.540	0.582	0.597	1.521	0.596	1.471
9-Circuit Network (Without Background Traffic)								
MINHOP	0.307	0.582	0.555	0.582	0.151	0.453	0.146	0.152
9-Circuit Network (With Background Traffic)								
MINHOP	0.495	0.831	0.562	0.917	0.223	0.290	0.218	0.312

seconds. One solution to the problem is to add a new circuit between WR-ALC and HQ AFLC. If a 9600 baud circuit is added, the maximum response times, with background traffic, drop to 1.748 seconds for the SM-ALC terminals and 0.602 seconds for the WR-ALC terminals. The average two way times were 1.057 seconds and 0.441 seconds respectively.

Summary

This chapter examined the connectivity, routing, negative acknowledgement and interactive terminal proposals using the BDN simulation model. It showed that block throughput can be increased by 30%, if the new 9600 baud SM-ALC to OO-ALC and OO-ALC to HQ AFLC circuits are added. Further, an additional 5-8% increase can be obtained when the current routing algorithm is replaced by the MINHOP routing algorithm

Negative acknowledgements were shown to be very useful in reducing lost time when an error in transmission occurs. In the current system, at the normal error rate of 1%, block throughput falls by 9.7% from the error free rate. When NAK's are used, block throughput falls by only 3%.

Finally, the model was used to obtain estimates for the average and maximum interactive packet delay times. Neither WR-ALC or SM-ALC had a maximum roundtrip response time below the two second limit set by AFLC. This situation can be remedied if a new WR-ALC to HQ AFLC circuit is added.

The next chapter concludes the study with a series of recommendations based on the results of this chapter.

VIII. Conclusion and Recommendations

This study was conducted in five steps. First, several performance characteristics and modeling techniques were evaluated to find the best method for studying the BDN. Once simulation had been chosen, the next step was to make a complete functional analysis of both the current network operation and the proposed changes. This analysis provided the requirements for the development of a simulation model. The model was developed using abstract data structures to represent the basic network mechanisms, nodes, circuits, tapes, and terminals, along with their associated control functions. Next, the model was verified and validated using the functional description and actual network performance data. Finally, the validated model was used to study the proposals developed by SKDAR.

Based on the results of the study, we can conclude that the additional circuits will improve network throughput by approximately 30% and thereby relieve the current congestion. However, to make maximum use of the additional capacity, we recommend that the minimum hop routing algorithm be adopted and that negative acknowledgement be added to the system. Neither proposal requires extensive software or any hardware changes. The routing change can be implemented by simply changing the routine that calculates the packet selection vectors for each circuit, to find the minimum path selections. The use of the vectors would remain the same. The NAK proposal can be implemented using the current status packet by reducing the CSN's from four bits to three bits and using the additional bit to signal an error. The MINHOP routing strategy will improve throughput and protect the network against packet looping even in a degraded mode of operation. The negative acknowledgements will minimize the impact of transmission error on message throughput.

We did not have enough information on the interactive terminal proposal to make a strong recommendation as to whether the updated network will provide the desired level of service. While it is clear that a maximum response time of less than two seconds cannot be obtained, the average times may be acceptable to AFLC. The single alternative explored in this study is not the only solution to the problem. The capacity of the existing circuits could be increased or alternative network configurations could be devised. SKDAR will have to develop the alternative solutions based on their knowledge of the system and long range objectives. The simulation model developed in this study could then be used to evaluate the proposals.

Overall, the thesis was very successful. The model developed in this study incorporates all of the significant features of the BDN, including:

1. Multi-site tape routing.
2. Variable length messages.
3. Transmission errors.
4. Batch traffic.
5. Interactive traffic.
6. Full node to node link protocols.
7. Full tape to tape message protocols.
8. BDN, MINHOP and MIN+1HOP routing algorithms.
9. Flow controls.

The selection of PASCAL as the programming language made the design and implementation of the model a relatively straight forward task. The key to the overall design was the use of abstract data types to represent basic network structures. This reduced the modeling process to simply linking the structures together. The final program uses much less main memory than other programs written in special purpose simulation languages. Based on this result, we recommend that this approach be used in similar network studies.

Bibliography

1. Davies, D. W., D. L. A. Barber, W. L. Price, and C. M. Solomonides, Computer Networks and Their Protocols, New York, N. Y.: John Wiley and Sons, Ltd., 1979.

2. Grogono, Peter, Programming in PASCAL, Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1980.

3. Healy, James W. and Denzel H. Henderson, Simulation and Analysis of AUTODIN II Network Design. MS Thesis. Wright-Patterson AFB, Ohio: Air Force Institute of Technology, December, 1980.

4. Horwitz, Ellis and Sartaj Sanhni, Fundamentals of Data Structures, Potomac, Maryland: Computer Science Press, Inc., 1976.

5. Kermani, P. and L. Kleinrock, "A Tradeoff Study of Switching Systems in Computer Communication Networks," IEEE Transactions On Computers, C-29: 1052-1060 (December 1980).

6. Kuo, Franklin, Protocols and Techniques for Data Communication Networks, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1981.

7. Meier, Robert C., Simulation in Business and Economics, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1969.

8. Norman, Robert, "AFIT Thesis Proposal," Conversation between Mr. Norman and Major Seward, February, 1981.

9. Norman, Robert, "Notes on the AFLC Bulk Data Network," unpublished (February 1981).

10. Ross, Douglas T., "Structured Analysis: A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January, 1977.

11. Seward, Walter, Engineering Professor, AFIT (Class Notes). Wright-Patterson AFB, Ohio, February, 1981.

12. Schoemarker, S., Computer Networks and Simulation. New York: North-Holland Publishing Company, 1978.

13. Schwartz, Misha, Computer-Communications Network Design and Analysis. New York: Prentice-Hall, Inc., 1978.

14. Sol, Inder M. and K. K. Aggarwal, "A Review of Computer-Communication Network Classification Schemes," IEEE Communications Magazine: 24-32 (March 1981).

APPENDIX A

Packet Layout

The BDN uses a single packet format for all types of packets. The length of a packet may range between 16 and 174 characters depending on type. Shorter packets are made by omitting unnecessary information. This section outlines the full packet format.

CHARACTERS	DESCRIPTION (EXAMPLE)
8	Crypto Syncs (211)
2	Line Syncs (026)
1	Field Defined in Bits
	1: Unused Not Transmitted (NT)
	1: Character Parity (Even)
	1: Overload Flag (Receive)
	1: Idle Flag (Transmit)
	1: Short ACK Flag
	4: Acknowledgement Channel Sequence Number (CSN)
1	Field Defined in Bits
	1: Unused (NT)
	1: Character Parity (Even)
	2: Security 00 = Unclassified 01 = Secret
	1: Line Check Flag
	.4: Output Channel Sequence Number (CSN)

CHARACTERS	DESCRIPTION (EXAMPLE)
1	Field Defined in Bits
	1: Unused (NT)
	1: Character Parity (Even)
	7: Destination(s)
1	Field Defined in Bits
	1: Unused (NT)
	1: Character Parity (Even)
	7: Source
2	Process Sequence Number (Block Number)
1	Packet Sequence Number
1	Total Number Packets in Message
1	Field Defined in Bits
	1: Unused (NT)
	1: Character Parity (Even)
	4: Destination Device Type
	3: Packet Priority
1	Control Parity (Even)
2	Start of Data and Control ACK's
2	Control ACK's
144	Data Characters
2	Parity for Data (Character Add)
1	ETX (203)
1	Packet Parity XOR Checksum
2	Push Characters
174	Total Characters

APPENDIX B

BDN Simulation User's Manual

This manual describes a three step process for modeling a computer network with the BDN Simulation Model. The process consists of: graphing the network, building specification tables, and making the data file. The manual concludes with a few comments on compiling and executing the program.

Network Layout

The first step is to graph the network as shown in Figure 31. Nodes are represented by a circle and given a number from 1 to N. The circuits are shown by lines connecting the nodes. Each circuit attached to a node is given a port number from 1 to P, where P is the number of circuits attached to the node. Next, each input tape is shown by a triangle. Finally, the terminals are shown in the same way using small squares.

Specification Tables

After the network has been laid out, the user should construct the three specification tables shown in Figures 32-34. The circuit table contains two rows for each full duplex circuit (one for each direction). The table

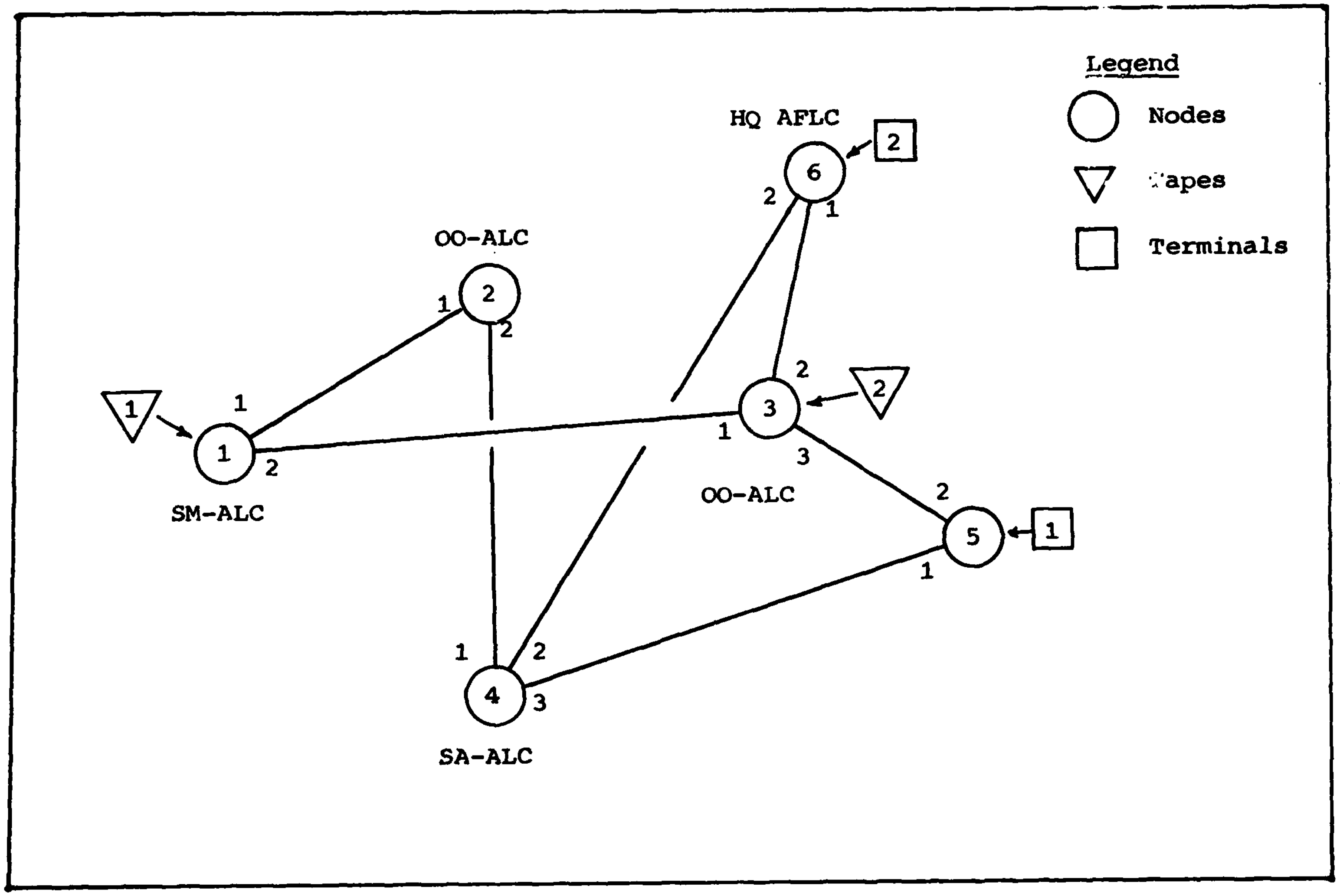

Figure 31. Sample Network Layout.

Circuit Table

Source		Destination		Capacity
Node	Port	Node	Port	(Bits/Second)
1	1	2	1	4800
	2	3	1	4800
2	1	1	1	4800
	2	4	1	4800
3	1	1	2	4800
	2	6	1	4800
	3	5	2	4800
4	1	2	2	4800
	2	6	2	4800
	3	5	1	4800
5	1	4	3	4800
	2	3	3	4800
6	1	3	2	4800
	2	4	2	4800

Figure 32. Sample Circuit Table.

Tape Table

Tape No.	Source	Destination(s)	Block Size Min.	Block Size Delta	Start Time
1	1	4	8	2	0
2	3	1,6	5	4	0

Figure 33. Sample Tape Table.

Terminal Table

Terminal No.	Source	Destination	Interarrival Parameters λ	Interarrival Parameters Min.	Interarrival Parameters Max.	Start Time
1	5	6	1.00	10	25000	0
2	6	5	0.75	10	25000	0

Figure 34. Sample Terminal Table.

identifies the return circuit's node and port numbers for each circuit. The tape table identifies the source, destination(s), block size parameters used to generate random message lengths (reference Equation 13, page 64) and starting time in milliseconds. The terminal table contains the source, destination, arrival rate, interarrival time boundary values, and terminal start time. This information is used to build the data file.

Data File

The user is now ready to build the data file. A sample input file is shown in Figure 35. The first line contains the number of nodes, number of tapes, number of terminals, transmission error rate, routing algorithm code (1 = BDN, 2 = MINHOP, and 3 = MIN+1HOP) and whether NAK's are to be used (1 = yes, 0 = no). The second line contains the stop time and interval between queue length statistic collections in seconds. The next N lines contain the name and number of active ports for each node. Next, the tapes are specified by copying the following information in the tape table: source node, destination(s) enclosed in square brackets, minimum block size, delta block size, and start time. The tape specifications are followed by the terminal specifications, which contain the source, destination enclosed in square brackets, arrival rate in seconds, minimum and maximum packet interarrival times in milliseconds

INPUT FILE

```
6 2 2 0.1 2 1
240  60
  SM-ALC    2
  OO-ALC    2
  OC-ALC    3
  SA-ALC    3
  WR-ALC    2
 HQ AFLC    2
1  [ 4 ]  8 2 0
3  [ 1 6 ] 5 4 0
5  [ 6 ] 1.00 10 25000 0
6  [ 5 ] 0.75 10 25000 0
2 1 4800
3 1 4800
1 1 4800
4 1 4800
1 2 4800
6 1 4800
5 2 4800
2 2 4800
6 2 4800
5 1 4800
4 3 4800
3 3 4800
3 2 4800
4 2 4800
```

Figure 35. Sample Input File.

and starting time in milliseconds. Finally, the network connectivity is given by specifying each circuit's return circuit (node and port) and capacity in bits per second.

Compiling and Executing the Program

The program can be compiled and run from a terminal using the following commands:

```
ATTACH, BDN, BDN SIMULATION (SOURCE FILE)
ATTACH, PASCAL, ID = AFIT
ATTACH, PASCLIB, ID = AFIT
ATTACH, PSRCLIB, ID = AFIT
EFL, 55000
PASCAL, BDN, LISTING, PROGRAM
PROGRAM, DATA, OUTPUT
```

Note: After the program has executed the output file can be displaced at the terminal and (or) routed to the line printer, using the command ROUTE, OUTPUT, DC = PR, TID = 91, FID = XXX, ST = CSB; where "XXX" is the output identifier.

SAMPLE OUTPUT

```
                         NETWORK CONFIGURATION
                         ---------------------

   NODE      PORT     DESTINATION     BAUDRATE     MINHOPS      MIN+1HOPS
==========================================================================

  SM-ALC      1         OO-ALC          4800       [010100]     [010100]

              2         OC-ALC          4800       [001011]     [001011]

  OO-ALC      1         SM-ALC          4800       [101000]     [101000]

              2         SA-ALC          4800       [000111]     [000111]

  OC-ALC      1         SM-ALC          4800       [110000]     [110000]

              2        HQ AFLC          4800       [000101]     [000101]

              3         WR-ALC          4800       [000110]     [000110]

  SA-ALC      1         OO-ALC          4800       [110000]     [110000]

              2        HQ AFLC          4800       [001001]     [001001]

              3         WR-ALC          4800       [001010]     [001010]

  WR-ALC      1         SA-ALC          4800       [010101]     [010101]

              2         OC-ALC          4800       [101001]     [101001]

 HQ AFLC      1         OC-ALC          4800       [101010]     [101010]

              2         SA-ALC          4800       [010110]     [010110]
lm +3
```

```
INTERIM JOBQUEUE LENGTHS
========================

TIME     NODE 1    NODE 2    NODE 3    NODE 4    NODE 5    NODE 6

  30       11         1        15         1         0         1

  60       15         1         8         1         0         5

  90       15         5        11         2         0         3

 120       10         6        11         3         1         3

 150       17         2        18         1         1         0

 180       10         9        11         1         0         2

 210        6         7        18         1         1         1

 240       12         3        13         0         0         2


 240       12         3        13         0         0         2

 STOPPED AT TIME:  240 SECONDS
```

```
TAPE RESULTS   ( TIMES IN MILLISECONDS )
=============  ========================

    TAPE NUMBER: 1      FROM: 1    TO: 4

 TYPE    NUMBER    MINIMUM    MAXIMUM    AVERAGE    UNITS/HOUR
===============================================================

BLOCK      66        3320       10711    7151.71      990.00

PACKET    565         582       10575    5337.51     8475.00

    TAPE NUMBER: 2      FROM: 3    TO: 1 6

 TYPE    NUMBER    MINIMUM    MAXIMUM    AVERAGE    UNITS/HOUR
===============================================================

BLOCK      71        2403       13037    6670.41     1065.00

PACKET    931         291       12902    3393.87    13965.00
TERMINAL RESULTS   ( TIMES IN MILLISECONDS )
--------------------------------------------


TERMINAL NO.: 1     FROM 5      TO: 6

NUMBER: 221     MINIMUM DELAY: 582     AVERAGE DELAY:    1133.66
MAXIMUM DELAY:  5290


TERMINAL NO.: 2     FROM 6      TO: 5

NUMBER: 168     MINIMUM DELAY: 582     AVERAGE DELAY:    1059.62
MAXIMUM DELAY:  3490
```

APPENDIX C

Program Listing

```
PROGRAM BDNSIMULATION (INPUT,OUTPUT);

(*$I'RANDOM'  RANDOM NUMBER GENERATOR DECLARATIONS.  *)

LABEL 999; (* ABORT *)

CONST

      MAXNODES = 6;
      NUMPORTS = 4;
      MAXTAPES = 18;
      MAXTERMINALS = 10;
      MAXQLENGTH = 99;
      HIGHESTPRIORITY = 3;
      SECOND = 1000;
      MAXTIME = 99999999;
      THRESHOLD = 71;
      MAXMESLENGTH = 20;
      MAKEPACKETTIME = 10;
      READBLOCKTIME = 500;
      BS = ' ';  (* BLANK SPACE *)
      CIRCUITOVERHEAD = 1; (* MILLISECOND *)

TYPE
     LOCATION = 1..MAXNODES;
     PORTS = 1..NUMPORTS;
     TAPES = 1..MAXTAPES;
     TERMINALS = 1 .. MAXTERMINALS;
     QUEUESIZE = 0..MAXQLENGTH;
     PRIORITY = 1..HIGHESTPRIORITY;
     CHANNELSEQUENCENUMBER = 0..7;
     DESTINATION = SET OF LOCATION;
     PACKET = ^PACKETS;
     QUEUE = PACKED RECORD
                        FRONT,REAR : PACKET;
                        LENGTH: QUEUESIZE;
               END;
     PRIQUEUE = RECORD
                    QUEUES: ARRAY[PRIORITY] OF QUEUE;
                    LENGTH: 0..99;
                  END;

     LINKEDLIST = PACKED RECORD
                             FRONT,REAR: PACKET;
                   END;
     PRIORITYLIST = PACKED ARRAY[PRIORITY] OF LINKEDLIST;
     JOBQUEUE = RECORD
                    PRILIST: PRIORITYLIST;
                    LENGTH: 0..90;
                  END;
     CIRCUITSTATUS = (IDLE,BEGINTRANSMISSION,ENDTRANSMISSION,
                        HOLD);
```

```
CIRCUITTYPE = RECORD
                 CIRCUITQ : QUEUE;
                 DESTNODE: LOCATION;
                 DESTPORT: PORTS;
                 NEXTCSN,EXPCSN: CHANNELSEQUENCENUMBER;
                 CURRENTPACKET: PACKET;
                 STATUS: CIRCUITSTATUS;
                 BAUDRATE: INTEGER;
                 ERRORDETECTED: BOOLEAN
              END;
NODES = RECORD
           NAME : ALFA;
           JOBQ: JOBQUEUE;
           PACKETIZER: PRIQUEUE;
           ACTIVEPORTS: 0..NUMPORTS;
           MINHOPS,MIN1HOPS: ARRAY[PORTS] OF DESTINATION;
        END;
NODEEVENTS = (PACKETREADY,STATUSUPDATE);

PUSHDOWNSTORE = RECORD
                   LIST: ARRAY [ LOCATION ] OF LOCATION;
                   TOP: 0 .. MAXNODES;
                END;
HOPSTABLE = ARRAY[LOCATION,PORTS] OF 0..99;
MESSAGESET = SET OF 1..MAXMESLENGTH;
SCTABLE = PACKED RECORD
                   NEXTBCN: 0..6000;
                   LASTPACKET: ARRAY[0..1] OF 0..MAXMESLENGTH;
                   UNACKSITES: ARRAY[0..1] OF DESTINATION;
                   TIMEIN: ARRAY[0..1] OF INTEGER;
          END;
RCTABLE = PACKED RECORD  (* RECEIVE CONTROL TABLE *)
                   EXPBCN: 0..6000;
                   UNACKPACKETS: ARRAY[0..1] OF MESSAGESET;
          END;
DATATYPE = (B,PDATA);
DELAYTABLE = ARRAY[DATATYPE,(NUM,TOTALTIME,MAX,MIN)] OF INTEGER;
TAPEDRIVE = RECORD
               FROM: LOCATION;
               DEST: DESTINATION;
               DT: DELAYTABLE;
               SCT: SCTABLE;
               RCT: ARRAY[LOCATION] OF RCTABLE;
               MINPKTS,DELTAPKTS: 0..MAXMESLENGTH;
               STARTTIME:INTEGER;
             END;

TERMINALTYPE = RECORD
                  FROM: LOCATION;
                  DEST: DESTINATION;
                  NUMBER: INTEGER;
                  TOTALDELAY,MINDELAY,MAXDELAY: INTEGER;
                  ARRIVALRATE: REAL;  (* UNITS/SECOND *)
                  MINWAITTIME,MAXWAITTIME: INTEGER;
```

```
                    END;

        PACKETTYPE = (DATA,ACK,STATUS,REP,TERMDATA);
        PACKETS = PACKED RECORD
                            FPTR,BPTR,QPTR: PACKET;
                            PRI: PRIORITY;
                            DESTSET: DESTINATION;
                            CSN,ACKCSN: CHANNELSEQUENCENUMBER;
                            FROM: LOCATION;
                            ONQUEUE: BOOLEAN;
                            PORTIN: 0..NUMPORTS;
                            NEGACK: BOOLEAN;
                            CASE TAG: PACKETTYPE OF
                              DATA,REP: (TAPENUMBER:TAPES;BCN:0..6000;PCN:1..15;MS
                                         :MESSAGESET;ENDOFMESSAGE:BOOLEAN;
                                         TIMEIN:INTEGER);
                              TERMDATA: ( TERMINALNUMBER: TERMINALS;
                                          TIMERECEIVED: INTEGER);
                              ACK: ();
                              STATUS: ();
                  END;

        EVENTS = (CIRCUITSTOP,TAPESTOPSEND,TAPESTOPREC,STOP,NODESTOP,
                  RECORDQLENGTHS,TERMINALSTOP);

    VAR
       NODE: ARRAY[LOCATION] OF NODES;
       NODETIME: ARRAY[LOCATION,NODEEVENTS] OF INTEGER;
       TAPE : ARRAY[TAPES] OF TAPEDRIVE;
       CIRCUIT : ARRAY[LOCATION,PORTS] OF CIRCUITTYPE;
       CIRCUITTIME : ARRAY[LOCATION,PORTS] OF INTEGER;
       TAPETIME: ARRAY[TAPES,LOCATION] OF INTEGER;
       TERMINAL : ARRAY[TERMINALS] OF TERMINALTYPE;
       TERMINALTIME: ARRAY[TERMINALS] OF INTEGER;
       MINTAPETIME : ARRAY[(TTIME,TNO,RECSITE)] OF INTEGER;
       TIME: INTEGER;
       SITE: LOCATION;
       PORT: PORTS;
       EVENT: EVENTS;
       NODESTATUS: NODEEVENTS;
       TDRIVE: TAPES;
       NUMNODES: LOCATION;
       NUMTAPES: TAPES;
       I,STOPTIME: INTEGER;
       NUMTERMINALS,TERM: 0..MAXTERMINALS;
       ERRORRATE: REAL;
       NODEEVENT: NODEEVENTS;
       STATTIME,DELTASTATTIME: INTEGER;
       CONNECTIVITY: ARRAY [ LOCATION,PORTS ] OF LOCATION;
       HOPS,INITHOPS: HOPSTABLE;
       ROUTINGALGORITHM: 1..3;      (* BDN = 1, MINHOP = 2,MIN+1 = 3 *)
       NAKSUSED: BOOLEAN;
       NAKFLAG: INTEGER;
```

```
(* -------------------------------------- *)

(* ---         INITIAL VALUES          --- *)

(* -------------------------------------- *)

    VALUE

    TIME = 0;
    NODE = (MAXNODES OF(' NODE NAME',JOBQUEUE(PRIORITYLIST(HIGHESTPRIORITY OF
             LINKEDLIST(NIL,NIL)),0),PRIQUEUE((HIGHESTPRIORITY OF (NIL,NIL,0)),0)
            ,0,(NUMPORTS OF []),(NUMPORTS OF [])));
    NODETIME = (MAXNODES OF (2 OF MAXTIME));
    TAPE = (MAXTAPES OF TAPEDRIVE(1,[],(2 OF(0,0,0,MAXTIME)),SCTABLE(2,(0,0),
            ([],[]),(0,0)),
            (MAXNODES OF RCTABLE(0,([1..MAXMESLENGTH],[1..MAXMESLENGTH]))),0,0,
              0));
    CIRCUIT = (MAXNODES OF (NUMPORTS OF CIRCUITTYPE((NIL,NIL,0),1,
              1,0,0,NIL,IDLE,0,FALSE)));
    CIRCUITTIME = (MAXNODES OF(NUMPORTS OF MAXTIME));
    TAPETIME = (MAXTAPES OF (MAXNODES OF MAXTIME));
    MINTAPETIME = (MAXTIME,1,1);
    TERMINAL = ( MAXTERMINALS OF (1,[],0,0,MAXTIME,0,0.0,0,0));
    TERMINALTIME = (MAXTERMINALS OF MAXTIME);
     INITHOPS = (MAXNODES OF(NUMPORTS OF 99));

(* --------------------------------------- *)

(* ---          QUEUE FUNCTIONS          --- *)

(* --------------------------------------- *)

PROCEDURE INITQUEUE (VAR Q:QUEUE);
(*  INITIALIZES A QUEUE  *)
BEGIN
   Q.FRONT := NIL;
   Q.LENGTH := 0;
END  (* INITQUEUE *);

FUNCTION QEMPTY (Q:QUEUE): BOOLEAN;
(*  IS THE QUEUE EMPTY?  *)
BEGIN
   QEMPTY := ( Q.FRONT=NIL )
END  (* QEMPTY *);

PROCEDURE ENQUEUE (VAR Q:QUEUE;VAR P:PACKET);
(*  ADDS A PACKET TO THE QUEUE  *)
BEGIN
   WITH Q DO
      BEGIN
         IF FRONT=NIL
            THEN
               BEGIN
                  FRONT := P;
```

```
                    REAR := P
                 END
              ELSE
                 BEGIN
                    REAR^.QPTR := P;
                    REAR := P
                 END;
           LENGTH := LENGTH + 1;
           P^.QPTR := NIL;
           P^.ONQUEUE := TRUE;
        END  (* WITH Q *)
END  (* ENQUEUE  *);

FUNCTION DEQUEUE (VAR Q:QUEUE): PACKET;
(*  RETURNS THE FIRST PACKET IN THE QUEUE  *)

VAR
    P: PACKET;
BEGIN
   WITH Q DO
      BEGIN
         IF LENGTH=0
            THEN
               BEGIN
                  WRITELN('  ERROR: ATTEMPT TO DEQUEUE AN EMPTY QUEUE');
                  GOTO 999
               END
            ELSE
               BEGIN
                  P := FRONT;
                  FRONT := FRONT^.QPTR;
                  P^.QPTR := NIL;
                  P^.ONQUEUE := FALSE;
                  DEQUEUE := P;
                  LENGTH := LENGTH - 1
               END;
      END  (* WITH Q  *)
END  (* DEQUEUE *);




(* --------------------------------------- *)

(* ---        PRIQUEUE FUNCTIONS        --- *)

(* --------------------------------------- *)

FUNCTION PQEMPTY (PQ:PRIQUEUE): BOOLEAN;
(*  RETURNS TRUE PRIORITY QUEUE IS EMPTY  *)
BEGIN
   PQEMPTY := (PQ.LENGTH = 0);
END;  (* PRIQUEUE EMPTY *)
```

```
PROCEDURE ENQUEUEPQ (VAR PQ:PRIQUEUE;P:PACKET);
(*  ADDS PACKET "P" TO PRIORITY QUEUE "PQ"  *)

VAR
    PRI: PRIORITY;
BEGIN
   PRI := P^.PRI;
   ENQUEUE(PQ.QUEUES[PRI],P);
   PQ.LENGTH := PQ.LENGTH + 1;
END;  (* ENQUEUE PRIQUEUE *)

FUNCTION DEQUEUEPQ (VAR PQ:PRIQUEUE): PACKET;
(*  RETURNS THE HIGHEST PRIORITY PACKET IN THE QUEUE  *)

VAR
    PRI: PRIORITY;
BEGIN
   IF PQEMPTY(PQ)
      THEN
         BEGIN
            WRITELN('  ERROR ATTEMPT TO DEQUEUE AN EMPTY PRIQUEUE');
            GOTO 999;
         END
      ELSE
         BEGIN
            PRI := HIGHESTPRIORITY ;
            WHILE QEMPTY(PQ.QUEUES[PRI]) DO
               PRI := PRI - 1;
            DEQUEUEPQ := DEQUEUE(PQ.QUEUES[PRI]);
            PQ.LENGTH := PQ.LENGTH - 1;
         END  (* ELSE *)
END;  (* DEQUEUE PRIQUEUE *)


(* --------------------------------------- *)

(* ---        JOBQUEUE FUNCTIONS        --- *)

(* --------------------------------------- *)

PROCEDURE INITJOBQUEUE (VAR JQ:JOBQUEUE);
(*  INITIALIZES A JOBQUEUE  *)

VAR
    PRI: PRIORITY;
BEGIN
   FOR PRI := 1 TO HIGHESTPRIORITY DO
      JQ.PRILIST[PRI].FRONT := NIL;
   JQ.LENGTH := 0
END  (* INITJOBQUEUE *);

FUNCTION JQEMPTY (JQ:JOBQUEUE): BOOLEAN;
(*  IS THE JOBQUEUE EMPTY?  *)
```

```
BEGIN
   JQEMPTY := ( JQ.LENGTH = 0 )
END  (* JQEMPTY *);

PROCEDURE ENQUEUEJQ (VAR JQ:JOBQUEUE;VAR P:PACKET);
(*  ADDS PACKET "P" TO JOB QUEUE "JQ"  *)

VAR
    PRI: PRIORITY;
BEGIN
   PRI := P^.PRI;
   WITH JQ.PRILIST[PRI] DO
      BEGIN
         IF FRONT = NIL
            THEN
               BEGIN
                  FRONT := P;
                  REAR := P;
                  P^.BPTR := NIL;
                  P^.FPTR := NIL;
               END
            ELSE
               BEGIN
                  REAR^.BPTR := P;
                  P^.FPTR := REAR;
                  REAR := P;
                  P^.BPTR := NIL;
               END;
      END  (* WITH JQ.PRILIST[PRI] *);
   JQ.LENGTH := JQ.LENGTH + 1
END  (* ENQUEUE *);

PROCEDURE DELETEPACKET (VAR JQ:JOBQUEUE;VAR P:PACKET);
(*  DELETES PACKET 'P' FROM THE JOBQUEUE  *)

VAR
    PRI: PRIORITY;
BEGIN
   PRI := P^.PRI;
   WITH JQ.PRILIST[PRI] DO
      BEGIN
         IF P = FRONT
            THEN
               FRONT := FRONT^.BPTR
            ELSE
               IF P^.BPTR = NIL
                  THEN
                     BEGIN
                        REAR := REAR^.FPTR;
                        REAR^.BPTR := NIL;
                     END
                  ELSE
                     BEGIN
                        P^.FPTR^.BPTR := P^.BPTR;
                        P^.BPTR^.FPTR := P^.FPTR;
                     END;
        END  (* WITH JQ.PRILIST[PRI] *);
     P^.BPTR := NIL;
     P^.FPTR := NIL;
     JQ.LENGTH := JQ.LENGTH - 1;
  END  (* DELETEPACKET *);

  (* ---------------------------------------- *)

  (* ---        STACK FUNCTIONS          --- *)

  (* ---------------------------------------- *)

  PROCEDURE INITSTACK (VAR STACK:PUSHDOWNSTORE);
  BEGIN
     STACK.TOP := 0;
  END;  (* INITIALIZE STACK *)

  FUNCTION STACKEMPTY (STACK:PUSHDOWNSTORE): BOOLEAN;
  (* TRUE IF STACK IS EMPTY *)
  BEGIN
     STACKEMPTY := STACK.TOP = 0
  END;  (* STACK EMPTY *)

  PROCEDURE PUSH ( VAR STACK:PUSHDOWNSTORE;SITE:LOCATION);
  (* ADDS SITE TO TOP OF STACK *)
  BEGIN
     STACK.TOP := STACK.TOP + 1;
     IF STACK.TOP > NUMNODES - 1
        THEN
           BEGIN
              WRITELN('  ERROR STACK OVERFLOW');
              GOTO 999;
           END
        ELSE
           STACK.LIST[STACK.TOP] := SITE;
  END;  (* PUSH *)

  FUNCTION POP ( VAR STACK:PUSHDOWNSTORE): LOCATION;
  (* RETURNS THE TOP LOCATION IN THE STACK *)
  BEGIN
     IF STACKEMPTY(STACK)
        THEN
           BEGIN
              WRITELN('  ERROR ATEMPT TO POP AN EMPTY STACK ');
              GOTO 999;
           END
        ELSE
           BEGIN
              POP := STACK.LIST[STACK.TOP];
              STACK.TOP := STACK.TOP - 1;
           END;
  END;  (* POP*)
```

```
(* ---------------------------------------- *)

(* ---        NODE VISITED FUNCTIONS    --- *)

(* ---------------------------------------- *)

FUNCTION ALLNODESVISITED(D:DESTINATION): BOOLEAN;
(* TRUE IF THE SET IS FULL *)
BEGIN
   ALLNODESVISITED := ( D = [1..NUMNODES] );
END;  (* ALL NODES VISITED *)

FUNCTION NODEVISITED ( D:DESTINATION;S:LOCATION): BOOLEAN;
(* TRUE IF SITE 'S' IS IN THE SET *)
BEGIN
   NODEVISITED := ( S IN D );
END;  (* NODE VISITED *)

PROCEDURE VISIT(VAR D:DESTINATION;SITE:LOCATION);
(* ADDS SITE TO THE SET OF DESTINATIONS VISITED *)
BEGIN
   D := D + [ SITE ];
END;  (* VISIT *)

(* ---------------------------------------- *)

(* ---    INITIALIZE ROUTING TABLES     --- *)

(* ---------------------------------------- *)

PROCEDURE INITBDNROUTINGTABLES;
(*  CALCULATES THE ROUTING SELECTION VECTORS FOR EACH
(*  CIRCUIT USING THE CURRENT BDN ROUTING ALGORITHM  *)

VAR
    SITE,STARTSITE,DEST: LOCATION;
    PORT,STARTPORT: PORTS;
    NODESVISITED: DESTINATION;
    STACK: PUSHDOWNSTORE;
BEGIN
   FOR STARTSITE := 1 TO NUMNODES DO
      WITH NODE[ STARTSITE ] DO
         FOR STARTPORT := 1 TO ACTIVEPORTS DO
            BEGIN
               INITSTACK(STACK);
               NODESVISITED := [STARTSITE];
               DEST := CONNECTIVITY[STARTSITE,STARTPORT];
               PUSH(STACK,DEST);
               VISIT(NODESVISITED,DEST);
               REPEAT
                  BEGIN
                     SITE := POP(STACK);
                     FOR PORT := 1 TO NODE[SITE].ACTIVEPORTS DO
                  BEGIN

                     DEST := CONNECTIVITY[SITE,PORT];
                     IF NOT NODEVISITED(NODESVISITED,DEST)
                        THEN
                         BEGIN
                          PUSH(STACK,DEST);
                          VISIT(NODESVISITED,DEST);
                         END;
                  END;
              END;
          UNTIL ALLNODESVISITED(NODESVISITED) OR STACKEMPTY(STACK);
          FOR PORT := 1 TO ACTIVEPORTS DO
             IF PORT <> STARTPORT
                THEN
                   NODESVISITED := NODESVISITED - [CIRCUIT[STARTSITE,PORT].
                                               DESTNODE];
          NODESVISITED := NODESVISITED - [ STARTSITE ];
          MINHOPS[STARTPORT] := NODESVISITED;
          MIN1HOPS[STARTPORT] := MINHOPS[STARTPORT];
       END;
END; (* INITIALIZE BDN ROUTING TABLES *)




(* --------------------------------------- *)

PROCEDURE FINDHOPSTABLE (VAR HOPS:HOPSTABLE;STARTSITE:LOCATION);
(* CALCULATES THE MINIMUM HOPS FROM CURRENT SITE TO ALL OTHER SITES *)

VAR
    SITE,DEST: LOCATION;
    PORT,STARTPORT: PORTS;
    NODESVISITED: DESTINATION;
    STACK: ARRAY [ 0..1 ] OF PUSHDOWNSTORE;
    DEPTH: INTEGER;
BEGIN
   HOPS := INITHOPS;
   WITH NODE[STARTSITE] DO
      FOR STARTPORT := 1 TO ACTIVEPORTS DO
         BEGIN
            INITSTACK(STACK[0]);
            INITSTACK(STACK[1]);
            NODESVISITED := [STARTSITE];
            HOPS[STARTSITE,STARTPORT] := 0;
            DEST := CONNECTIVITY[STARTSITE,STARTPORT];
            DEPTH := 1;
            HOPS[DEST,STARTPORT] := DEPTH;
            PUSH(STACK[0],DEST);
            VISIT(NODESVISITED,DEST);
            DEPTH := DEPTH + 1;
            REPEAT
```

```
                  BEGIN
                     SITE := POP(STACK[(DEPTH ) MOD 2]);
                     FOR PORT := 1 TO NODE[SITE].ACTIVEPORTS DO
                        BEGIN
                           DEST := CONNECTIVITY[SITE,PORT];
                           IF NOT NODEVISITED(NODESVISITED,DEST)
                              THEN
                                 BEGIN
                                  PUSH(STACK[(DEPTH + 1 ) MOD 2],DEST);
                                  VISIT(NODESVISITED,DEST);
                                  HOPS[DEST,STARTPORT] := DEPTH;
                                 END;
                        END;
                     IF STACKEMPTY( STACK[ DEPTH MOD 2 ])
                        THEN
                           DEPTH := DEPTH + 1;
                  END
               UNTIL ALLNODESVISITED(NODESVISITED) OR
                     STACKEMPTY(STACK[(DEPTH ) MOD 2]);
            END (* FOR *)
END;  (* FIND HOPS TABLE *)

PROCEDURE INITMINHOPROUTING;
(* INITIALIZE MINHOP OR MIN+1HOP ROUTING TABLES *)

VAR
    SITE,STARTSITE: LOCATION;
    PORT: PORTS;
    MINSTEPS : ARRAY[LOCATION] OF 0..99;
BEGIN
   FOR STARTSITE := 1 TO NUMNODES DO
      BEGIN
         FINDHOPSTABLE(HOPS,STARTSITE);
         FOR SITE := 1 TO NUMNODES DO
            BEGIN
               MINSTEPS[SITE] := 99;
               FOR PORT := 1 TO NODE[STARTSITE].ACTIVEPORTS DO
                  IF MINSTEPS[SITE] > HOPS[SITE,PORT]
                     THEN
                        MINSTEPS[SITE] := HOPS[SITE,PORT];
            END;
         WITH NODE[STARTSITE] DO
            BEGIN
               FOR PORT := 1 TO ACTIVEPORTS DO
                  FOR SITE := 1 TO NUMNODES DO
                     IF ( (HOPS[SITE,PORT] = MINSTEPS[SITE]) AND (SITE <>
                        STARTSITE)
                        AND (MINSTEPS[SITE] < 99) )
                        THEN
                           BEGIN
                              MINHOPS[PORT] := MINHOPS[PORT] + [SITE];
                              MIN1HOPS[PORT] := MIN1HOPS[PORT] + [SITE];
                           END
                        ELSE
```

```
                              IF ( (ROUTINGALGORITHM = 3) AND
                                 ((HOPS[SITE,PORT] - 1) = MINSTEPS[SITE]) AND
                                 (SITE <> STARTSITE) )
                                 THEN
                                  MIN1HOPS[PORT] := MIN1HOPS[PORT] + [SITE];
               END;
           END (* FOR *);
       END;  (* INITIALIZE MINHOP ROUTING TABLES *)



       (* ---------------------------------------- *)

       (* ---       ROUTING ALGORITHM          --- *)

       (* ---------------------------------------- *)



       FUNCTION FINDNEWPACKET ( SITE: LOCATION; PORT:PORTS): PACKET;
       (*  RETURNS THE FIRST PACKET IN THE JOB QUEUE WHICH CAN BE
           OUTPUT OVER THE REQUESTING CIRCUIT  *)

       VAR
           PRI: 0..HIGHESTPRIORITY;
           P: PACKET;
           PACKETFOUND: BOOLEAN;

        FUNCTION REACHABLE: BOOLEAN;
          BEGIN
             WITH NODE[SITE] DO
                IF P^.FROM = SITE
                   THEN
                      REACHABLE := ((P^.DESTSET * MIN1HOPS[PORT]) <> [])
                   ELSE
                      REACHABLE := ((P^.DESTSET * MINHOPS[PORT]) <> []);
          END;

       BEGIN
          PACKETFOUND := FALSE;
          PRI := HIGHESTPRIORITY;
          REPEAT
             BEGIN
                P := NODE[SITE].JOBQ.PRILIST[PRI].FRONT;
                WHILE (P<>NIL) AND NOT PACKETFOUND DO
                   BEGIN
                      IF REACHABLE AND NOT (P^.ONQUEUE)
                             AND NOT ( P^.PORTIN = PORT )
                         THEN
                            PACKETFOUND := TRUE
                         ELSE
                            P := P^.BPTR;
                   END;
                PRI := PRI - 1;
```

```
      END;
   UNTIL PACKETFOUND OR (PRI =0);
   FINDNEWPACKET := P;
END  (* FINDNEWPACKET *);

(* ---------------------------------------- *)

(* ---          TERMINAL FUNCTIONS       --- *)

(* ---------------------------------------- *)

FUNCTION NEXTARRIVALTIME(VAR T:TERMINALS): INTEGER;
(*  RETURNS THE ARRIVAL TIME FOR THE NEXT TERMINAL PACKET
    USING POISSON ARRIVALS  *)

VAR
    NEXTTIME: INTEGER;
BEGIN
   WITH TERMINAL[T] DO
      BEGIN
         NEXTTIME := -( ROUND( (1000/ARRIVALRATE) * LN(RAN) ));
         IF NEXTTIME < MINWAITTIME
            THEN
               NEXTTIME := MINWAITTIME
            ELSE
               IF NEXTTIME > MAXWAITTIME
                  THEN
                     NEXTTIME := MAXWAITTIME;
      END;
   NEXTARRIVALTIME := TIME + NEXTTIME;
END;  (* NEXT ARRIVAL TIME *)

PROCEDURE SENDTERMPACKET(T:TERMINALS);
(*  GENERATES A TERMINAL DATA PACKET AND PLACES IT IN THE
    SENDING NODES PACKETIZER  *)

VAR
    P: PACKET;
BEGIN
   WITH TERMINAL[T] DO
      BEGIN
         NEW(P);
         P^.TAG := TERMDATA;
         P^.FROM := FROM;
         P^.DESTSET := DEST;
         P^.PRI := 2;
         P^.ONQUEUE := FALSE;
         P^.TERMINALNUMBER := T;
         P^.TIMERECEIVED := TIME;
         P^.FPTR := NIL;
         P^.BPTR := NIL;
         P^.QPTR := NIL;
         ENQUEUEPQ(NODE[FROM].PACKETIZER,P);
         NODETIME[FROM,PACKETREADY] := TIME;
```

```
            NUMBER := NUMBER + 1;
         END;  (* WITH TERMINAL *)
      TERMINALTIME[T] := NEXTARRIVALTIME(T);
   END;  (* SEND TERMINAL PACKET *)

   PROCEDURE PROCESSTERMDATA ( P:PACKET );
   (*  RECORDS THE DELAY TIME FOR THE INCOMMING TERMINAL PACKET *)

   VAR
       DELAY: INTEGER;
   BEGIN
      WITH TERMINAL[P^.TERMINALNUMBER] DO
          BEGIN
             DELAY := TIME - P^.TIMERECEIVED;
             TOTALDELAY := TOTALDELAY + DELAY;
             IF DELAY < MINDELAY
                THEN
                   MINDELAY := DELAY
                ELSE
                   IF DELAY > MAXDELAY
                      THEN
                         MAXDELAY := DELAY;
          END;
   END;  (* PROCESS TERMINAL DATA PACKET *)

   (* ---------------------------------------- *)

   (* ---            NODE FUNCTIONS         --- *)

   (* ---------------------------------------- *)

   PROCEDURE CHECKCIRCUITS(N:LOCATION;P:PACKET);
   (*  CHECKS TO SEE IF THERE IS AN IDLE CIRCUIT WHICH CAN TRANSMIT
       THE NEW PACKET, IF SO, THE CIRCUIT IS RESTARTED  *)

   VAR
       PORT: PORTS;
       INTERSECTION: DESTINATION;
   BEGIN
      FOR PORT := 1 TO NODE[N].ACTIVEPORTS DO
          BEGIN
             IF P^.FROM = SITE
                THEN
                   INTERSECTION := ( NODE[N].MIN1HOPS[PORT] * P^.DESTSET )
                ELSE
                   INTERSECTION := ( NODE[N].MINHOPS[PORT] * P^.DESTSET );
             IF ((CIRCUIT[N,PORT].STATUS = IDLE) AND
                (INTERSECTION <> []))
                THEN
                   BEGIN
                      CIRCUIT[N,PORT].STATUS := BEGINTRANSMISSION;
                      CIRCUITTIME[N,PORT] := TIME + CIRCUITOVERHEAD;
                   END;
          END;
```

```
END;



PROCEDURE FINDMINTAPETIME;
(* FIND THE TIME FOR THE NEXT TAPE EVENT AND STORES IT FOR
   USE BY THE FIND NEXT EVENT FUNCTION  *)

VAR
    T: TAPES;
    L: LOCATION;
BEGIN
   MINTAPETIME[TTIME] := MAXTIME;
   FOR T := 1 TO NUMTAPES DO
      FOR L := 1 TO NUMNODES DO
         IF TAPETIME[T,L] < MINTAPETIME[TTIME]
            THEN
               BEGIN
                  MINTAPETIME[TTIME] := TAPETIME[T,L];
                  MINTAPETIME[TNO] := T;
                  MINTAPETIME[RECSITE] := L;
               END;
END;  (* FIND MINIMUM TAPE TIME *)

PROCEDURE PROCESSNODESTOP (N:LOCATION;NODESTATUS:NODEEVENTS);
(*  PROCESSES THE TWO NODE EVENTS  *)

 VAR P: PACKET;
     PORT: PORTS;
     ENDOFMESSAGE: BOOLEAN;
 BEGIN
    WITH NODE[N] DO
       CASE NODESTATUS OF
          PACKETREADY: BEGIN  (* MOVE A PACKET TO THE JOB QUEUE  *)
                          ENDOFMESSAGE := FALSE;
                          IF JOBQ.LENGTH >= THRESHOLD
                             THEN
                                NODETIME[N,PACKETREADY] := MAXTIME
                             ELSE
                                BEGIN
                                 P := DEQUEUEPQ(PACKETIZER);
                                 P^.PORTIN := 0;
                                 P^.TIMEIN := TIME;
                                 ENQUEUEJQ(JOBQ,P);
                                  CHECKCIRCUITS(N,P);
                                 IF P^.TAG = DATA
                                  THEN
                                   IF P^.ENDOFMESSAGE
                                    THEN
                                     BEGIN
                                      ENDOFMESSAGE := TRUE;
                                      TAPETIME[P^.TAPENUMBER,N] := TIME+ 30 *
                                                                 SECOND;
```

```
                                                     FINDMINTAPETIME;
                                                    END;
                                                IF PQEMPTY(PACKETIZER)
                                                 THEN
                                                  NODETIME[N,PACKETREADY] := MAXTIME
                                                 ELSE
                                                  IF ENDOFMESSAGE
                                                   THEN
                                                    NODETIME[N,PACKETREADY] := TIME +
                                                                           READBLOCKTIME
                                                   ELSE
                                                    NODETIME[N,PACKETREADY] := TIME +
                                                                           MAKEPACKETTIME;
                                              END;  (* ELSE *)
                                           END; (* PACKET READY *)
                 STATUSUPDATE: FOR PORT := 1 TO ACTIVEPORTS DO
                                          BEGIN
                                             (* GENERATE A STATUS PACKET FOR EACH
                                               OUTPUT CIRCUIT *)
                                             NEW(P);
                                             P^.TAG := STATUS;
                                             P^.FROM := N;
                                             P^.ONQUEUE := FALSE;
                                             P^.PRI := 3;
                                             P^.DESTSET :=
                                                          [ CIRCUIT[N,PORT].DESTNODE];
                                             ENQUEUEPQ(PACKETIZER,P);
                                             NODETIME[N,STATUSUPDATE] := TIME + 12
                                                                       * SECOND;
                                             PROCESSNODESTOP(N,PACKETREADY);
                                          END;
            END;
      END;

      PROCEDURE PROCSTATUSPACKET(P:PACKET);
      (*  DUMMY OPERATION USED TO PROCESS INCOMMING STATUS PACKET  *)
      BEGIN
         P^.ONQUEUE := FALSE
      END;

      (* ---------------------------------------- *)

      (* ---          MESSAGE FUNCTIONS        --- *)

      (* ---------------------------------------- *)

      FUNCTION MESSAGELENGTH (MINPKTS,DELTAPKTS: INTEGER): INTEGER;
      (*  RETURNS THE NUMBER OF PACKETS IN THE NEXT BLOCK  *)
      BEGIN
         MESSAGELENGTH := TRUNC( MINPKTS + RAN * DELTAPKTS);
      END;

      FUNCTION BCNVALID (I:INTEGER): BOOLEAN;
      (*  DETERMINES IF THE BCN OF THE INCOMMING PACKET IS
IN RANGE  *)
BEGIN
   BCNVALID := ( (I=0) OR (I=1) )
END;


PROCEDURE RECORDTIMES (VAR DT:DELAYTABLE;DTYPE:DATATYPE;DELAY:INTEGER);
(*  RECORDS THE DELAY TIME OF THE ARRIVING DATA PACKET  *)

BEGIN
   DT[DTYPE,NUM] := DT[DTYPE,NUM] + 1;
   DT[DTYPE,TOTALTIME] := DT[DTYPE,TOTALTIME] + DELAY;
   IF DELAY < DT[DTYPE,MIN]
      THEN
         DT[DTYPE,MIN] := DELAY;
   IF DELAY > DT[DTYPE,MAX]
      THEN
         DT[DTYPE,MAX] := DELAY;
END;  (* RECORD DELAY TIMES *)



PROCEDURE GENREPLYPACKET(DEST:DESTINATION;MS:MESSAGESET;SITE:LOCATION;
                              BCN:INTEGER;T:TAPES);
(*  GENERATES A REPLY PACKET TO ACKNOWLEDGE A BLOCK OR REQUEST
    RETRANSMISSION OF THE MISSING PACKETS  *)

VAR
    RP: PACKET;
BEGIN
   NEW(RP);
   RP^.PRI := 3;
   RP^.FROM := SITE;
   RP^.TAG := REP;
   RP^.BCN := BCN;
   RP^.DESTSET := DEST;
   RP^.MS := MS;
   RP^.ONQUEUE := FALSE;
   RP^.TAPENUMBER := T;
   ENQUEUEPQ(NODE[SITE].PACKETIZER,RP);
   NODETIME[SITE,PACKETREADY] := TIME;
END;  (* GENERATE REPLY PACKET *)

PROCEDURE PROCESSRECTIMEOUT (T:TAPES;SITE:LOCATION);
(*  REQUESTS RETRANSMISSION OF THE MISSING PACKETS WHEN THE
    RCT 2 SECOND TIMER RUNS OUT  *)

BEGIN
   WITH TAPE[T] DO
      WITH RCT[SITE] DO
         BEGIN
            GENREPLYPACKET([FROM],UNACKPACKETS[0],SITE,EXPBCN,T);
         END;
   TAPETIME[T,SITE] := MAXTIME;
   FINDMINTAPETIME;
END;  (* PROCESS RECEIVE TIMEOUT *)

PROCEDURE GENERATEPACKETS(TAPENO,BCN,LASTPACKETNO,FROM,PRI:INTEGER;
                          DEST:DESTINATION;MS:MESSAGESET);
(*  GENERATES A BLOCK OF DATA PACKETS AND PLACES THEM IN THE
    PACKETIZER AT THE SOURCE NODE  *)

VAR
    I: INTEGER;
    P: PACKET;
BEGIN
   FOR I := 1 TO LASTPACKETNO DO
      IF I IN MS
         THEN
            BEGIN
               NEW(P);
               P^.TAG := DATA;
               P^.TAPENUMBER := TAPENO;
               P^.QPTR := NIL;
               P^.ONQUEUE := FALSE;
               P^.PCN := I;
               P^.BCN := BCN;
               P^.PRI := PRI;
               P^.DESTSET := DEST;
               P^.MS := [];
               P^.FROM := FROM;
               P^.TIMEIN := TIME;
               IF I = LASTPACKETNO
                  THEN
                     P^.ENDOFMESSAGE := TRUE
                  ELSE
                     P^.ENDOFMESSAGE := FALSE;
               ENQUEUEPQ(NODE[FROM].PACKETIZER,P);
            END  (* IF I IN MS *);
   IF NODETIME[FROM,PACKETREADY] = MAXTIME
      THEN
         NODETIME[FROM,PACKETREADY] := TIME + MAKEPACKETTIME;
END;  (* GENERATE DATA PACKETS *)

PROCEDURE RESETTIMER (T:TAPES;N:NODES);
BEGIN
         TAPETIME[T,TAPE[T].FROM] := MAXTIME;
   FINDMINTAPETIME;
END;  (* RESET SEND TIMER *)

PROCEDURE PROCESSREPLY ( P:PACKET);
(*   RETRANSMITS THE MISSING PACKETS OR SENDS THE NEXT DATA
    BLOCK DEPENDING ON THE NATURE OF THE REPLY  *)

VAR
    I: INTEGER;
BEGIN
   WITH TAPE[P^.TAPENUMBER] DO
```

```
        BEGIN
           I := P^.BCN - SCT.NEXTBCN + 2;
           IF BCNVALID(I)
              THEN
                 BEGIN
                    IF P^.MS = []
                       THEN
                          SCT.UNACKSITES[I] := SCT.UNACKSITES[I] - [P^.FROM]
                       ELSE
                          BEGIN
                             GENERATEPACKETS(P^.TAPENUMBER,P^.BCN,
                                             SCT.LASTPACKET[I],
                                             FROM,1,
                                             [P^.FROM],P^.MS);
                             RESETTIMER(P^.TAPENUMBER,NODE[FROM]);
                          END;
                    WHILE SCT.UNACKSITES[0] = [] DO
                       WITH SCT DO
                          BEGIN
                             RECORDTIMES(DT,B,(TIME-TIMEIN[0]));
                             UNACKSITES[0] := UNACKSITES[1];
                             TIMEIN[0] := TIMEIN[1];
                             LASTPACKET[0] := LASTPACKET[1];
                             UNACKSITES[1] := DEST;
                             TIMEIN[1] := TIME;
                             LASTPACKET[1] := MESSAGELENGTH(MINPKTS,DELTAPKTS);
                             GENERATEPACKETS(P^.TAPENUMBER,NEXTBCN,
                                LASTPACKET[1],FROM,1,DEST,[1..LASTPACKET[1]]);
                             RESETTIMER(P^.TAPENUMBER,NODE[FROM]);
                             NEXTBCN := NEXTBCN + 1;
                          END  (* WITH SCT *);
                 END  (* IF *)
        END  (* END WITH TAPE *);
   END;  (* PROCESS REPLY PACKET *)

   PROCEDURE PROCESSDATAMESSAGE( P:PACKET;SITE:LOCATION);
   (*  RECORDS THE PCN OF THE ARRIVING PACKET, ACKNOWLEDGES THE BLOCK
       WHEN ALL THE PACKETS HAVE ARRIVED AND SETS UP A NEW
       RECEIVE-CONTROL-TABLE FOR THE NEXT BLOCK.  *)

   VAR
       I: INTEGER;
   BEGIN
      WITH TAPE[P^.TAPENUMBER].RCT[SITE] DO
         BEGIN
            I := P^.BCN - EXPBCN;
             IF BCNVALID(I)
                THEN
                   IF (P^.PCN IN UNACKPACKETS[I])
                      THEN
                         BEGIN
                            UNACKPACKETS[I] := UNACKPACKETS[I] - [P^.PCN];
                            IF P^.ENDOFMESSAGE
                               THEN
```

```
                                            BEGIN
                                             UNACKPACKETS[I] := UNACKPACKETS[I]
                                                               -[(P^.PCN + 1)..MAXMESLENGTH]
                                             ;
                                             IF (I = 0) AND (UNACKPACKETS[0] <> [])
                                              THEN
                                               BEGIN
                                                TAPETIME[P^.TAPENUMBER,SITE] := TIME+ 2 *
                                                                               SECOND;
                                                FINDMINTAPETIME;
                                               END
                                            END;  (* IF ENDOFMESSAGE *)
                                      WHILE  (UNACKPACKETS[0] = []) DO
                                         BEGIN
                                            GENREPLYPACKET([P^.FROM],[],SITE,EXPBCN,
                                                           P^.TAPENUMBER);
                                            TAPETIME[P^.TAPENUMBER,SITE] := MAXTIME;
                                            FINDMINTAPETIME;
                                            UNACKPACKETS[0] := UNACKPACKETS[1];
                                            UNACKPACKETS[1] := [1..MAXMESLENGTH];
                                            EXPBCN := EXPBCN + 1;
                                         END;
                                   END  (* IF BCNVALID *);
                    END  (* WITH TAPE *)
              END;  (* PROCESS DATA MESSAGE *)

              PROCEDURE RETRANSMITBLOCK (T: TAPES);
              (*  RETRANSMITS THE LAST TWO BLOCKS WHEN THE 30 SECOND TIMER RUNS OUT  *)

              VAR
                  I: INTEGER;
              BEGIN
                 WITH TAPE[T] DO
                    WITH SCT DO
                       BEGIN
                          FOR I := 0 TO 1 DO
                             BEGIN
                                IF LASTPACKET[I] = 0
                                   THEN
                                      BEGIN
                                         LASTPACKET[I] := MESSAGELENGTH(MINPKTS,DELTAPKTS);
                                         TIMEIN[I] := TIME;
                                         UNACKSITES[I] := DEST;
                                      END;
                                GENERATEPACKETS(T,(NEXTBCN -2 +I),LASTPACKET[I],FROM,1,
                                DEST,[ 1..LASTPACKET[I]]);
                                RESETTIMER(T,NODE[FROM]);
                             END; (* FOR *)
                       END;
              END;  (* RETRANSMIT BLOCK *)
```

```
(* ---------------------------------------- *)

(* ---          CIRCUIT FUNCTIONS        --- *)

(* ---------------------------------------- *)

FUNCTION '.OADED(JQ:JOBQUEUE): BOOLEAN;
BEGIN
   LOADED := ( JQ.LENGTH >= THRESHOLD );
END;



FUNCTION TRANSMISSIONTIME(PT:PACKETTYPE;BAUDRATE:INTEGER): INTEGER;
(*  CALCULATES THE TRANSMISSION TIME FOR THE PACKET  *)

BEGIN
   CASE PT OF
      DATA,TERMDATA: TRANSMISSIONTIME := ROUND((8*174*1000)/BAUDRATE);
      ACK,REP,STATUS: TRANSMISSIONTIME := ROUND((8*16*1000)/BAUDRATE);
   END  (* CASE *);
END  (* TRANSMISSIONTIME *);


PROCEDURE GENERATEACK (RETURNSITE:LOCATION;RETURNPORT:PORTS);
(*  GENERATES A SHORT ACK PACKET AT THE END OF A BURST
IF THE RETURN CIRCUIT IS IDLE  *)

VAR
    P: PACKET;
BEGIN
   WITH CIRCUIT[RETURNSITE,RETURNPORT] DO
      BEGIN
         IF STATUS = IDLE
            THEN
               BEGIN
                  NEW(P);
                  P^.TAG := ACK;
                  P^.ACKCSN := EXPCSN;
                  P^.NEGACK := ERRORDETECTED;
                  ERRORDETECTED := FALSE;
                  P^.QPTR := NIL;
                  CURRENTPACKET := P;
                  STATUS := ENDTRANSMISSION;
                  CIRCUITTIME[RETURNSITE,RETURNPORT] := TIME +
                                                        TRANSMISSIONTIME(ACK
                                                        ,
                                                        BAUDRATE
                                                        );
               END;
      END  (* WITH CIRCUIT[] *);
END  (* GENERATEACK *);

PROCEDURE PROCESSSTARTTRANSMISSION (SITE:LOCATION;PORT:PORTS);
```

```
 VAR
     BURSTSIZE: INTEGER;

BEGIN  (* PROCESS START TRANSMISSION *)
   WITH CIRCUIT[SITE,PORT] DO
       BEGIN
           IF LOADED(NODE[DESTNODE].JOBQ)
              THEN
                 BURSTSIZE := 1
              ELSE
                 BURSTSIZE := 4;
           IF (CURRENTPACKET = NIL) AND (CIRCUITQ.LENGTH >= BURSTSIZE)
              THEN
                 BEGIN
                    STATUS := HOLD;
                    CIRCUITTIME[SITE,PORT] := TIME + 1 * SECOND;
                    GENERATEACK(DESTNODE,DESTPORT);
                 END
              ELSE
                 IF CURRENTPACKET <> NIL
                    THEN  (* RETRANSMIT *)
                       BEGIN
                          CURRENTPACKET^.ACKCSN := EXPCSN;
                          CURRENTPACKET^.NEGACK := ERRORDETECTED;
                          ERRORDETECTED := FALSE;
                          STATUS := ENDTRANSMISSION;
                          CIRCUITTIME[SITE,PORT] := TIME + TRANSMISSIONTIME(
                                                    CURRENTPACKET^.TAG,
                                                    BAUDRATE);
                       END
                    ELSE
                       BEGIN
                          CURRENTPACKET := FINDNEWPACKET(SITE,PORT);
                          IF (CURRENTPACKET = NIL) AND ( QEMPTY(CIRCUITQ) )
                             THEN
                                BEGIN
                                 STATUS := IDLE;
                                 CIRCUITTIME[SITE,PORT] := MAXTIME;
                                END
                             ELSE
                                IF CURRENTPACKET = NIL
                                 THEN
                                  BEGIN
                                    STATUS := HOLD;
                                    CIRCUITTIME[SITE,PORT] := TIME + 1 * SECOND;
                                   GENERATEACK(DESTNODE,DESTPORT);
                                  END
                                 ELSE  (* START A NEW TRANSMISSION *)
                                  BEGIN
                                   ENQUEUE(CIRCUITQ,CURRENTPACKET);
                                   CURRENTPACKET^.CSN := NEXTCSN;
                                   NEXTCSN := ( NEXTCSN +1) MOD 8;
                                   CURRENTPACKET^.ACKCSN := EXPCSN;
```

```
                                         CURRENTPACKET^.NEGACK := ERRORDETECTED;
                                         ERRORDETECTED := FALSE;
                                         STATUS := ENDTRANSMISSION;
                                         CIRCUITTIME[SITE,PORT] := TIME +
                                                                 TRANSMISSIONTIME
                                                                 (CURRENTPACKET^.TAG,
                                                                 BAUDRATE);
                                        END
                               END
                   END  (* WITH CIRCUIT[] *);
                END  (* STARTTRANSMISSION *);

(* ---------------------------------------- *)

PROCEDURE PROCESSHOLD(SITE:LOCATION;PORT:PORTS);
(*  RETRANSMITS A BURST OF PACKETS WHEN THE 1 SECOND SENT AND WAIT
    TIMER RUNS OUT  *)

BEGIN
   WITH CIRCUIT[SITE,PORT] DO
      BEGIN
         CURRENTPACKET := CIRCUITQ.FRONT;
         STATUS := BEGINTRANSMISSION;
      END;  (* WITH CIRCUIT[] *)
   PROCESSSTARTTRANSMISSION(SITE,PORT);
END;  (* PROCESSHOLD *)

(* ---------------------------------------- *)

FUNCTION ERROR : BOOLEAN;
(*  GENERATES RANDON TRANSMISSION ERRORS  *)
BEGIN
   ERROR := (RANDOM < ERRORRATE);
END;

FUNCTION CLEARCIRCUIT(SITE:LOCATION;PORT:PORTS): PACKET;
(*  CLEARS THE TRANSMITTED PACKET OUT OF THE SENDING END OF THE CIRCUIT
    AND RETURNS A COPY OF THE PACKET WITH THE APPROPRIATE DOWN LINE
    DESTINATIONS SET  *)

VAR
    P: PACKET;
    INTERSECTION: DESTINATION;
BEGIN
   WITH CIRCUIT[SITE,PORT] DO
      BEGIN
         NEW(P);
         P^ := CURRENTPACKET^;
         IF P^.FROM = SITE
            THEN
               INTERSECTION := ( NODE[SITE].MIN1HOPS[PORT] * P^.DESTSET )
            ELSE
               INTERSECTION := ( NODE[SITE].MINHOPS[PORT] * P^.DESTSET );
         P^.DESTSET := INTERSECTION;
```

```
            P^.QPTR := NIL;
            P^.FPTR := NIL;
            P^.BPTR := NIL;
            P^.ONQUEUE := FALSE;
            CLEARCIRCUIT := P;
            IF P^.TAG = ACK
               THEN
                  DISPOSE(CURRENTPACKET)
               ELSE
                  CURRENTPACKET := CURRENTPACKET^.QPTR;
            STATUS := BEGINTRANSMISSION;
            CIRCUITTIME[SITE,PORT] := TIME + CIRCUITOVERHEAD;
         END  (* WITH CIRCUIT[] *)
END;  (* CLEARCIRCUIT *)


PROCEDURE PROCESSACK (P:PACKET;SITE:LOCATION;PORT:PORTS);
(*  RELEASES THE ACKNOWLEDGED PACKETS FROM THE CIRCUIT QUEUE  *)

VAR
    GO,RESTART: BOOLEAN;
    INTERSECTION: DESTINATION;
    TEMP: PACKET;

FUNCTION ACKVALID : BOOLEAN;
(*  RETURNS TRUE IF THE ACK IS FOR PACKETS IN THE CIRCUIT QUEUE  *)

VAR
    ACK,F,R: CHANNELSEQUENCENUMBERS;
BEGIN
   IF QEMPTY(CIRCUIT[SITE,PORT].CIRCUITQ)
      THEN
         ACKVALID := FALSE
      ELSE
         BEGIN
            F := CIRCUIT[SITE,PORT].CIRCUITQ.FRONT^.CSN;
            R := CIRCUIT[SITE,PORT].CIRCUITQ.REAR^.CSN;
            ACK := (P^.ACKCSN + 7) MOD 8;  (* ACKCSN - 1 *)
            IF F <= R
               THEN
                  ACKVALID := ((F <= ACK) AND (ACK <= R))
               ELSE
                  ACKVALID :=  ((F <= ACK) OR (ACK <= R));
         END;  (* ELSE *)
END  (* ACKVALID *);

BEGIN (* PROCESSACK *)
   WITH CIRCUIT[SITE,PORT] DO
      BEGIN
         IF ACKVALID
            THEN
               BEGIN
                  RESTART := FALSE;
                  GO := TRUE;
```

```
                    WHILE GO DO
                       BEGIN
                          TEMP := CIRCUITQ.FRONT;
                          IF CURRENTPACKET = TEMP
                             THEN
                                BEGIN
                                 RESTART := TRUE;
                                 CURRENTPACKET := CURRENTPACKET^.QPTR;
                                END;
                          TEMP := DEQUEUE(CIRCUITQ);
                          IF TEMP^.FROM = SITE
                             THEN
                                INTERSECTION := ( TEMP^.DESTSET * NODE[SITE].
                                                MIN1HOPS[PORT] )
                             ELSE
                                INTERSECTION := ( TEMP^.DESTSET * NODE[SITE].
                                                MINHOPS[PORT] );
                          IF ( TEMP^.DESTSET - INTERSECTION ) <> []
                             THEN
                                BEGIN
                                 TEMP^.DESTSET := TEMP^.DESTSET - INTERSECTION
                                END
                             ELSE
                                BEGIN
                                 WITH NODE[SITE] DO
                                  BEGIN
                                   DELETEPACKET(JOBQ,TEMP);
                                   DISPOSE(TEMP);
                                   IF NOT PQEMPTY(PACKETIZER) AND (JOBQ.LENGTH <
                                      THRESHOLD) AND
                                      (NODETIME[SITE,PACKETREADY] = MAXTIME)
                                    THEN
                                     NODETIME[SITE,PACKETREADY] := TIME;
                                  END (* WITH NODE[] *);
                                END;
                          IF QEMPTY(CIRCUITQ)
                             THEN
                                GO := FALSE
                             ELSE
                                IF (P^.ACKCSN = CIRCUITQ.FRONT^.CSN)
                                 THEN
                                  GO := FALSE;
                       END;
                    IF NOT P^.NEGACK AND ((STATUS = HOLD) OR RESTART)
                       THEN
                          PROCESSSTARTTRANSMISSION(SITE,PORT);
                 END;
           IF P^.NEGACK
              THEN
                 BEGIN
                    CURRENTPACKET := CIRCUITQ.FRONT;
                    PROCESSSTARTTRANSMISSION(SITE,PORT);
                 END;
        END;
```

```
END  (* PROCESSACK *);



PROCEDURF PROCESSENDTRANSMISSION(SITE:LOCATION;PORT:PORTS);
(*  PROCESS THE INCOMING PACKET  *)

VAR
    P: PACKET;
    DN: LOCATION;
    DP: PORTS;

FUNCTION INORDER (EXPCSN:CHANNELSEQUENCENUMBER): BOOLEAN;
(*  RETURNS TRUE IF THE PACKET IS THE EXPECTED PACKET  *)
BEGIN
   IF P^.TAG = ACK
      THEN
         INORDER := FALSE
      ELSE
         INORDER := (P^.CSN = EXPCSN)
END;

FUNCTION NODEFULL: BOOLEAN;
BEGIN
   NODEFULL := (NODE[DN].JOBQ.LENGTH >= 90)
END;

PROCEDURE PROCESSMESSAGE(P:PACKET;SITE:LOCATION);
BEGIN
   CASE P^.TAG OF
      DATA: PROCESSDATAMESSAGE(P,SITE);
      REP:  PROCESSREPLY(P);
      TERMDATA:  PROCESSTERMDATA(P);
      STATUS: PROCSTATUSPACKET(P);
   END;  (* CASE *)
END;  (* PROCESS MESSAGE *)


BEGIN   (* PROCESS END TRANSMISSION *)
   DN := CIRCUIT[SITE,PORT].DESTNODE;
   DP := CIRCUIT[SITE,PORT].DESTPORT;
   P := CLEARCIRCUIT(SITE,PORT);
   WITH CIRCUIT[DN,DP] DO
      BEGIN
            IF ERROR THEN
              IF NAKSUSED THEN
                BEGIN
                   ERRORDETECTED := TRUE;
                   DISPOSE(P);
                   GENERATEACK(DN,DP);
                END
              ELSE DISPOSE(P)
              ELSE
                 BEGIN
                    PROCESSACK(P,DN,DP);
                    IF INORDER(EXPCSN) AND NOT NODEFULL
                       THEN
                          BEGIN
                             EXPCSN := (EXPCSN + 1) MOD 8;
                             IF (DN IN P^.DESTSET)
                                THEN
                                 PROCESSMESSAGE(P,DN);
                             P^.DESTSET := P^.DESTSET - [DN];
                             IF P^.DESTSET <> []
                                THEN
                                 BEGIN
                                  P^.PORTIN := DP;
                                  ENQUEUEJQ(NODE[DN].JOBQ,P);
                                  CHECKCIRCUITS(DN,P);
                                 END
                                ELSE
                                 BEGIN
                                  IF P^.TAG = DATA
                                   THEN
                                    RECORDTIMES(TAPE[P^.TAPENUMBER].DT,PDATA,
                                                (TIME - P^.TIMEIN));
                                  DISPOSE(P);
                                 END
                          END
                       ELSE
                          DISPOSE(P);
                 END
        END  (* WITH *);
     END (* PROCESS RECEIVE TRANSMISSION *);

  (* ---------------------------------------- *)

  FUNCTION FINDNEXTEVENT(VAR TIME:INTEGER): EVENTS;
  (*  RETURNS THE NEXT EVENT ALONG WITH THE APPROPRIATE INDICES
      NEEDED TO PROCESS THE EVENT  *)

  VAR
      I,J: INTEGER;
      NE: NODEEVENTS;
  BEGIN
     TIME := MAXTIME;
     IF TIME > MINTAPETIME[TTIME]
        THEN
           BEGIN
              TIME := MINTAPETIME[TTIME];
              IF MINTAPETIME[RECSITE] = TAPE[MINTAPETIME[TNO]].FROM
                 THEN
                    FINDNEXTEVENT := TAPESTOPSEND
                 ELSE
                    FINDNEXTEVENT := TAPESTOPREC;
           END;
      FOR I := 1 TO NUMNODES DO
         FOR J := 1 TO NODE[I].ACTIVEPORTS DO
            IF TIME > CIRCUITTIME[I,J]
               THEN
                  BEGIN
                     FINDNEXTEVENT := CIRCUITSTOP;
                     TIME := CIRCUITTIME[I,J];
                     SITE := I;
                     PORT := J;
                  END;

      FOR I := 1 TO NUMNODES DO
         FOR NE := PACKETREADY TO STATUSUPDATE DO
            IF TIME > NODETIME[I,NE]
               THEN
                  BEGIN
                     TIME := NODETIME[I,NE];
                     SITE := I;
                     FINDNEXTEVENT := NODESTOP;
                     NODEEVENT := NE;
                  END;

      FOR I := 1 TO NUMTERMINALS DO
         IF TIME > TERMINALTIME[I]
            THEN
               BEGIN
                  TIME := TERMINALTIME[I];
                  TERM := I;
                  FINDNEXTEVENT := TERMINALSTOP;
               END;

      IF TIME > STATTIME
         THEN
            BEGIN
               TIME := STATTIME;
               FINDNEXTEVENT := RECORDQLENGTHS;
            END;
      IF TIME > STOPTIME
         THEN
            BEGIN
               TIME := STOPTIME;
               FINDNEXTEVENT := STOP;
            END;
   END;  (* FIND NEXT EVENT *)

   (* ---------------------------------------- *)

   (* ---     INPUT/OUTPUT FUNCTIONS       --- *)

   (* ---------------------------------------- *)

   PROCEDURE READDESTSET(VAR D:DESTINATION);
   (*  READS A LIST OF LOCATIONS ENCLOSED IN BRACKES AND STORES THEM
       IN A DESTINATION SET  *)
```

```
VAR
    CH: CHAR;
    L: LOCATION;
BEGIN
   D := [];
   WHILE INPUT^ <> '[' DO
      READ(CH);
   READ(CH);  (* READ '[' *)
   WHILE ( INPUT^ <> ']' ) AND ( NOT EOLN ) DO
      BEGIN
         READ(L);
         D := D + [L];
         READ(CH);
      END;
   IF INPUT^ = ']'
      THEN
         READ(CH);
END;  (* READ DESTINATION SET *)

PROCEDURE READALFA(VAR A:ALFA);

VAR
    CH: CHAR;
    I: INTEGER;
BEGIN
   FOR I := 1 TO 10 DO
      BEGIN
         READ(CH);
         A[I] := CH;
      END;
END;  (* READ ALFA VARIABLE *)


PROCEDURE PRINTNETCONFIGURATION;
(*  PRINTS THE NETWORK CONNECTIVITY INFORMATION AND ROUTING
    VECTORS IN A TABLE  *)

VAR
    L: LOCATION;
    PORT: PORTS;
    I: INTEGER;
BEGIN
   WRITELN;
   WRITELN;
   WRITELN(BS:22,'NETWORK CONFIGURATION');
   WRITELN(BS:22,'---------------------');
   WRITELN;
   WRITELN(BS:8,'NODE',BS:5,'PORT',BS:5,'DESTINATION',BS:5,'BAUDRATE',
           BS:5,'MINHOPS',BS:5,'MIN+1HOPS');
   WRITELN(BS:5,'==================================================',
           '=====================');
   WRITELN;
   FOR L:= 1 TO NUMNODES DO
      FOR PORT := 1 TO NODE[L].ACTIVEPORTS DO
         BEGIN
            IF PORT = 1
               THEN
                  WRITE(BS:5,NODE[L].NAME:10)
               ELSE
                  WRITE(BS:15);
            WITH CIRCUIT[L,PORT] DO
               BEGIN
                  WRITE(PORT:4,BS:8,NODE[DESTNODE].NAME:10,BAUDRATE:11);
                  WRITE(BS:7,'[');
                  FOR I := 1 TO NUMNODES DO
                     IF I IN NODE[L].MINHOPS[PORT]
                        THEN
                           WRITE('1')
                        ELSE
                           WRITE('0');
                  WRITE(']');
                  WRITE('    [');
                  FOR I := 1 TO NUMNODES DO
                     IF I IN NODE[L].MIN1HOPS[PORT]
                        THEN
                           WRITE('1')
                        ELSE
                           WRITE('0');
                  WRITELN(']');
                  WRITELN;
               END;  (* CIRCUIT[] *)
         END;  (* FOR *)
END;  (* PRINT NETWORK CONFIGURATION *)




PROCEDURE PRINTTAPETIMES(T: TAPES);

VAR
    L: LOCATION;
BEGIN
   WITH TAPE[T] DO
      BEGIN
         WRITELN;
         WRITE(BS:10,'TAPE NUMBER:',T:2,BS:5,'FROM: ',
               FROM:1,BS:3,'TO:');
         FOR L := 1 TO NUMNODES DO
             IF L IN DEST
                THEN
                   WRITE(L:2);
         WRITELN;
         WRITELN;
         WRITELN(BS:6,'TYPE    NUMBER    MINIMUM    MAXIMUM    ',
                 'AVERAGE    UNITS/HOUR');
```

```
          WRITELN(BS:5,'=====================================',
                       '=====================');
          WRITELN;
          WRITELN(BS:5,'BLOCK ',DT[B,NUM]:7,DT[B,MIN]:12,DT[B,MAX]:12,
                  (DT[B,TOTALTIME]/DT[B,NUM]): 11: 2,
                 (DT[B,NUM]*3600/((TIME - STARTTIME)/1000)):12:2);
          WRITELN;
          WRITELN(BS:5,'PACKET',DT[PDATA,NUM]:7,DT[PDATA,MIN]:12,
                   DT[PDATA,MAX]:12,
                   (DT[PDATA,TOTALTIME]/DT[PDATA,NUM]): 11: 2,
                                                   (DT[PDATA,NUM]*3600/(
                                 (TIME - STARTTIME)/1000 )):12:2);
           WRITELN;
       END;  (* WITH TAPE *)
END;  (* PRINT TAPE TIMES *)

PROCEDURE PRINTQLENGTHHEADING;

VAR
    I: INTEGER;
BEGIN
   WRITELN('1    INTERIM JOBQUEUE LENGTHS');
   WRITELN('     =======================');
   WRITELN;
   WRITE('    TIME   ');
   FOR I := 1 TO NUMNODES DO
      WRITE('  NODE ',I:1,'  ');
   WRITELN;
   WRITELN;
END;  (* PRINT HEADING FOR JOBQUEUE LENGTHS *)

PROCEDURE PROCRECORDQLENGTHS;

VAR
    SITE: LOCATION;
BEGIN
   WRITE(' ',(TIME DIV 1000): 8,'  ');
   FOR SITE := 1 TO NUMNODES DO
      WRITE(NODE[SITE].JOBQ.LENGTH:6,BS:4);
   WRITELN;
   WRITELN;
   STATTIME := TIME + DELTASTATTIME;
END;  (* PROCESS RECORD JOBQUEUE LENGTHS *)

PROCEDURE PRINTTERMINALRESULTS;
VAR
   T: 0..MAXTERMINALS; D: LOCATION;
BEGIN
   WRITELN('1    TERMINAL RESULTS   ( TIMES IN MILLISECONDS )');
   WRITELN('     -----------------------------------------');
   WRITELN;WRITELN;
   FOR T := 1 TO NUMTERMINALS DO
     WITH TERMINAL[T] DO
       BEGIN
```

```
         WRITE(BS:5,'TERMINAL NO.:',T:2,BS:5,'FROM',FROM:2,BS:5);
         D := 1; WHILE NOT (D IN DEST) DO D := D + 1;
         WRITELN('TO:',D:2);WRITELN;
         WRITELN(BS:5,'NUMBER:',NUMBER:4,BS:5,'MINIMUM DELAY:',MINDELAY:4,
                 BS:5,'AVERAGE DELAY:',(TOTALDELAY/NUMBER):10:2,
                 BS:5,'MAXIMUM DELAY:',MAXDELAY:6);WRITELN;WRITELN;
   END
END;  (* PRINT TERMINAL RESULTS *)

(* ---------------------------------------- *)

(* ---       BEGIN MAIN PROGRAM         --- *)

(* ---------------------------------------- *)

BEGIN
   SETRANDOM(3,3);  (* INITIALIZE RANDOM NUMBER GENERATORS *)
   SETRAN(3);
   READLN(NUMNODES,NUMTAPES,NUMTERMINALS,ERRORRATE,ROUTINGALGORITHM,
          NAKFLAG);
   NAKSUSED := (NAKFLAG = 1);
   READLN(STOPTIME,DELTASTATTIME);
   (*  CONVERT TIMES TO MILLISECONDS *)
   STOPTIME := STOPTIME * SECOND;
   DELTASTATTIME := DELTASTATTIME * SECOND;
   STATTIME := DELTASTATTIME;
   FOR SITE := 1 TO NUMNODES DO
      BEGIN
         READALFA(NODE[SITE].NAME);
         READLN(NODE[SITE].ACTIVEPORTS);
      END;
   FOR TDRIVE := 1 TO NUMTAPES DO
    WITH TAPE[TDRIVE] DO
      BEGIN
        READ(FROM);
        READDESTSET(DEST);
        READLN(MINPKTS,DELTAPKTS,STARTTIME);
        TAPETIME[TDRIVE,FROM] := STARTTIME;
      END;  (* FOR TDRIVE *)
   FOR TERM := 1 TO NUMTERMINALS DO
      WITH TERMINAL[TERM] DO
         BEGIN
           READ(FROM);
           READDESTSET(DEST);
            READLN(ARRIVALRATE,MINWAITTIME,MAXWAITTIME,TERMINALTIME[TERM]);
         END;
   FOR SITE := 1 TO NUMNODES DO
      WITH NODE[SITE] DO
         FOR PORT := 1 TO ACTIVEPORTS DO
            WITH CIRCUIT[SITE,PORT] DO
               BEGIN
                  READ(DESTNODE,DESTPORT);
                    READ(BAUDRATE);
                  READLN;
```

```
                    CONNECTIVITY[SITE,PORT] := DESTNODE;
                 END;
     CASE ROUTINGALGORITHM OF
        1: INITBDNROUTING;
        2,3: INITMINHOPROUTING;
     END;
     PRINTNETCONFIGURATION;
     PRINTQLENGTHHEADING;
     FOR TDRIVE := 1 TO NUMNODES DO
        NODETIME[TDRIVE,STATUSUPDATE] := TDRIVE * 2 * SECOND;
     FINDMINTAPETIME;
     EVENT := FINDNEXTEVENT(TIME);
     WHILE EVENT <> STOP DO
        BEGIN
           CASE EVENT OF
              CIRCUITSTOP: CASE CIRCUIT[SITE,PORT].STATUS OF
                              BEGINTRANSMISSION: PROCESSSTARTTRANSMISSION(
                                                    SITE,PORT);
                              HOLD: PROCESSHOLD(SITE,PORT);
                              ENDTRANSMISSION: PROCESSENDTRANSMISSION(
                                                            SITE,PORT);
                           END;
              TAPESTOPREC: PROCESSRECTIMEOUT(MINTAPETIME[TNO],MINTAPETIME[
                                             RECSITE]);
              TAPESTOPSEND: RETRANSMITBLOCK(MINTAPETIME[TNO]);
              NODESTOP: PROCESSNODESTOP(SITE,NODEEVENT);
              TERMINALSTOP: SENDTERMPACKET(TERM);
              RECORDQLENGTHS: PROCRECORDQLENGTHS;
           END;
           EVENT := FINDNEXTEVENT(TIME);
        END;
     WRITELN;
     PROCRECORDQLENGTHS;
     WRITELN('     STOPPED AT TIME: ',(TIME DIV 1000): 4,' SECONDS');
     WRITELN('1      TAPE RESULTS   ( TIMES IN MILLISECONDS )');
     WRITELN('       ==================================');
     FOR TDRIVE := 1 TO NUMTAPES DO
        PRINTTAPETIMES(TDRIVE);
     PRINTTERMINALRESULTS;
     999:
  END.
*EOR
```

VITA

Stephen D. Stewart was born on 22 November 1949 in Miami, Florida. He graduated from Southwest Miami Senior High School in 1968 and attended the University of Florida in Gainesville, Florida from which he received the degree Bachelor of Science in Industrial Engineering in August 1972. Mr. Stewart began his career as a USAF civilian employee in September of that year. He enrolled in Georgia College, Milledgeville, Georgia, in 1974 from which he received the degree Master of Business Administration in August 1975. He entered the School of Engineering, Air Force Institute of Technology, in June 1980. He is a member of Tau Beta Pi.

Permanent address: 103 Wendell Court
Warner Robins, Georgia 31093

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/81D-15	2. GOVT ACCESSION NO. AD-A115 552	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SIMULATION AND ANALYSIS OF THE AFLC BULK DATA NETWORK USING ABSTRACT DATA TYPES		5. TYPE OF REPORT & PERIOD COVERED MS Thesis
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Stephen D. Stewart Civ USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS RAIDS & Program Shop AFCCPC/SKDAR Tinker AFB, OK 73145		12. REPORT DATE December, 1981
		13. NUMBER OF PAGES 156
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

15 APR 1982

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

~~FREDRIC C. LYNCH, Maj, USAF~~ [illegible]
~~Director of Public Affairs~~

Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Simulation, Packet Switching and Modeling.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The AFLC Bulk Data Network (BDN) is a packet switching network which electronically transfers bulk computer data (magnetic tapes) between the six AFLC bases. The network operates at "full" capacity for extended periods of time at the beginning of each month. This results in undesirable delays in transferring data between the sites. To remedy the problem AFCCPC/SKDAR plans to add an additional circuit to the network and increase the transmission capacity of another circuit. (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A computer simulation was developed and validated to analyze this and other proposals formulated to improve network performance.

The simulation makes extensive use of abstract data types to represent key network components. The abstract data structures provide a set of functional building blocks which are linked together to model a variety of network confingurations. The program incorporates full node to node link and tape to tape message protocols. The model can be used to study both interactive and batch traffic, using one of three different routing algorithms.